HZ BOOKS
华章IT

WebRTC标准和SIP标准主要作者联袂撰写，互联网实时通信技术的权威指南，Amazon全五星评价

全方位深入解读标准的方方面面，阐释WebRTC相关概念、技术及最佳实践，包含大量的演示应用、示例代码、各类数据和图表

Web开发技术丛书

WebRTC

APIs and RTCWEB Protocols of the HTML5 Real-Time Web, Third Edition

[美] 艾伦 B. 约翰斯顿（Alan B. Johnston）
丹尼尔 C. 伯内特（Daniel C. Burnett） 著
声网Agora.io 译

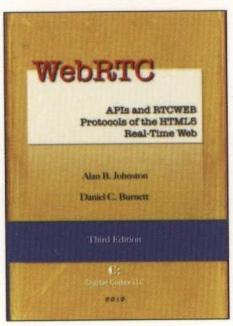
WebRTC 权威指南

（原书第3版）

机械工业出版社
China Machine Press

本书由WebRTC标准和SIP标准的主要作者联袂撰写，全方位深入解读标准的方方面面，并通过大量的演示应用、示例代码、各类数据和图表，深入浅出地阐释WebRTC相关概念、技术及最佳实践，为互联网实时通信开发者和技术决策者提供权威参考。

全书共14章，第1章简要介绍WebRTC及其新的功能特性；第2章介绍如何建立WebRTC会话、会话运行期间可执行哪些操作，以及如何关闭会话等；第3章介绍WebRTC的媒体模型以及如何捕获和控制本地媒体；第4章介绍信令相关知识；第5章介绍如何建立对等媒体；第6章介绍对等连接和提议/应答协商；第7章介绍数据通道、JavaScript API以及底层协议；第8章介绍W3C WebRTC标准文档；第9章介绍NAT和防火墙穿透；第10章讨论与WebRTC相关的协议；第11章介绍IETF文档；第12章介绍与IETF相关的RFC文档；第13章介绍和探讨WebRTC安全和隐私的诸多方面；第14章介绍主流浏览器对WebRTC API和协议的支持。



原书封面

Web开发技术丛书

WebRTC
权威指南

（原书第3版）

[美] 艾伦 B. 约翰斯顿（Alan B. Johnston）
丹尼尔 C. 伯内特（Daniel C. Burnett） 著
声网Agora.io 译

WebRTC

APIs and RTCWEB Protocols of the HTML5 Real-Time Web, Third Edition

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

WebRTC 权威指南（原书第 3 版）/（美）艾伦 B. 约翰斯顿（Alan B. Johnston），（美）丹尼尔 C. 伯内特（Daniel C. Burnett）著；声网 Agora.io 译．—北京：机械工业出版社，2016.8
（Web 开发技术丛书）
书名原文：WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web, Third Edition

ISBN 978-7-111-54715-0

I. W… II. ①艾… ②丹… ③声… III. 移动终端 - 应用程序 - 程序设计 IV. TN929.53

中国版本图书馆 CIP 数据核字（2016）第 199660 号

本书版权登记号：图字：01-2016-5123

WebRTC 权威指南（原书第 3 版）

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：陈佳媛　　责任校对：董纪丽
印　　刷：三河市宏图印务有限公司　　版　　次：2016 年 9 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：16.25
书　　号：ISBN 978-7-111-54715-0　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

Preface 中文版序言

中国的互联网产业正以爆炸式的速度增长。云服务、开源技术、HTML5 以及移动端 SDK 让中国的开发者们迅速地建立起面向全世界的网页端和移动端应用。互联网 + 创业也已经成为国家级的主题，这代表了科技给中国带来的新机会。WebRTC 正是其中极富潜力的一个。自 2011 年 Google 发起 WebRTC 开源项目和标准化工作以来，WebRTC 已经成为未来最有希望统一互联网音视频通信服务的技术标准。尽管初衷是建立浏览器之间的音视频通信能力，但作为一个高质量的开源音视频引擎，WebRTC 也帮助了成千上万的开发者和项目团队为移动应用和其他常见使用搭建通信功能。这一点进一步扩大了 WebRTC 在全行业的影响力，以及未来的发展空间。

声网（Agora.io）很高兴能在中国推出这一领域最有影响力的技术书籍《WebRTC 权威指南》。作者 Daniel C. Burnett 博士是 WebRTC 标准的主要作者，在书中对标准的方方面面做了精确到位的介绍。Alan B. Johnston 博士则是今日通信业核心标准 SIP 的主要作者，多年的行业实践和全局视野让他能够深入浅出地给出 WebRTC 相关技术问题和发展方向的真知灼见。加上书中大量详实的具体解读、演示应用、示例代码以及各类数据和图表，本书已成为互联网实时通信开发者和技术决策者最权威的参考文献，英文版已经再版多次。

作为对 WebRTC 社区的贡献，声网希望此书的中文版能够填补这方面中文资料的空白，帮助中国软件开发者和技术团队了解和使用这一开源项目和技术标准，在时代的科技大潮中大展宏图。

赵斌

声网 CEO

Agora.io

第 3 版序言 *Preface*

WebRTC 领域的变化是惊人的，但有时是反复无常的。有时，它迅速前进，取得重大进展。有时，它看起来很慢，几乎像冰川一样一动不动，毫无进展。当今的互操作性是一流的，并使得我们的演示应用程序在浏览器（Chrome 和 Firefox）和平台（Windows 和 Mac）上实现语音、视频和数据通信变成一件轻而易举的事。而另一方面，关于潜在视频互操作性问题和其他的 API 还有一些疑惑。

时间会告诉我们哪些人物会主宰 WebRTC 向前发展。当然，相比之前的第 1 版，标准和实现的日趋完善及融合，让我们的代码编写和写作变得更加得心应手。

这个版本的新内容体现在增强的演示应用程序，它展示了如何实现浏览器之间直接发送实时文本的数据通道功能。此外，还涉及浏览器媒体协商过程中的完整描述（Firefox 和 Chrome 的 SDP 会话描述），如何使用 Wireshark 来监控 WebRTC 协议的注意事项以及例子捕捉。另外，支持 NAT 和防火墙穿透的 TURN 服务器也是本版新加入的内容。

此版本还逐步介绍了 WebRTC，阐述了诸如本地媒体、信令等概念，并通过独立可运行的演示程序来介绍对等连接。与往常一样，所有的代码都可以在 http://webrtcbook.com/code3.html 免费下载，你还可以在 http://demo.webrtcbook.com 上试用。[⊖]我们希望这个新版本能有助于你实现 WebRTC 的开发和集成。

⊖ 本书相关资源也可登录华章网站（www.hzbook.com）下载。——编辑注

Preface 第 2 版序言

自从出版了本书第 1 版，在几个月的时间里，WebRTC 继续得到演化和发展。

虽然在 IETF 和 W3C 标准的许多领域已经取得了实质性的进展，但我们还有很多工作要做。事件触发，流在协议层次上的表示，甚至连一些回调函数的语法也还在讨论的话题之内；例如多个视频轨道如何体现在单个媒体流里，当一个绑定在 HTML 元素的媒体流出现音视频轨道的新增或者删除时会发生什么，这些细节都才刚开始进入考虑范畴。但是，API 的核心在逐渐稳固。在使用方面，由于人们力求理解 WebRTC 的影响和机遇，很多会议、聚会网站、创业公司如雨后春笋般涌现，而且规模得到快速扩张。

其他趋势也变得清晰明了。有关编解码器的分歧，尤其是视频编解码器，正从“打斗”“战争”慢慢发展到了“冷战”。笔者衷心希望在标准和行业上能就视频编码器的实现迅速达成一致。

虽然现在 WebRTC 的基本组成部分在演示和应用程序上运行良好，但是围绕着安全和信令通道（这在本书第 2 版有全新的章节进行介绍），WebRTC 还有许多顾虑和问题。此外，现在网络上的示例代码要么过于复杂，要么解释不充分，因此这也促使本版呈现出完整运行且充分解释的例子。这个能在 Chrome 和 Firefox 浏览器上运行的演示代码，也能在网站 http://demo.webrtcbook.com 上下载。

我们希望更新后的版本能激起你对 WebRTC 的兴趣并了解 WebRTC。祝你阅读愉快！

第 1 版序言 *Preface*

互联网和万维网已经改变了我们的世界。当撰写这段历史时，大多会谈到 20 世纪末和 21 世纪初这些技术对生活的影响。网络改变了我们获取信息、与其他人互动、工作和娱乐的方式。现在，网络即将极大地改变我们使用语音和视频沟通的方式。这本书给出了被称为 WebRTC（Web Real-Time Communications）的最新的标准和行业成就最新的快照。这项技术与 HTML5 浏览器的其他进展，拥有可能彻底地改变我们在私人和商业领域沟通方式的潜力。

笔者多年涉足互联网通信行业，见证了互联网上语音和视频通信的进步和影响。我们致力于制定信令协议，例如会话发起协议（SIP）和会话描述协议（SDP），以及安全协议，例如用于语音和视频通信系统的 ZRTP，它将为势必替代电话系统的公共交换电话网络或 PSTN 奠定基础。这些互联网通信技术带来了一波惊人的“破坏”浪潮，但我们相信 WebRTC 很有可能造成更大的“破坏”。

这本书为想要抓住这个新浪潮（尽管它还在形成中）的 Web 开发人员和电话开发人员提供了信息。WebRTC 所需的标准和协议仍在开发和发明中。浏览器也开始渐渐地支持实现 WebRTC 功能。然而，作者认为需要有一本书来解释这项仍在发展中的技术。本书将解释 WebRTC 的技术目标、结构、协议和应用程序编程接口（API）。我们计划频繁地更新本书的版本，可能一年会有三次，并且倾向于数字化交付和按需出版以节省成本并获得最大超链接有效性。想要获得最新版本的信息和更新及修改的列表，请访问 http://webrtcbook.com。

本书首先介绍了 WebRTC，进而讨论了它的新进展。我们研究了 WebRTC 对等媒体流的独特方面，并且解释了网络地址转换（NAT）穿越。然后讨论了 W3C 和 IETF 组织的工作文件，最终确定了那些正在形成 WebRTC 标准的文件。每章都以参考资料结尾，并列出各种标准文件。以 [RFC ...] 形式出现的引用是 IETF 请求注解文件，以 [draft- .....] 形式出现

的引用是 IETF 标准草案工作文件，里面的内容可能会因本书的出版而被更新或者修改。在大多数情况下，本书所提供的超链接能检索到最新版本的文档。W3C 草案的引用包括最新公开工作草案的链接和最新编辑草案的链接。对于那些不熟悉 W3C 和 IETF 标准化进程的读者，我们提供了附录 A 和附录 B 作为参考。最后我们讨论了目前 WebRTC 在主流浏览器中的部署状况。

如果你是一名 Web 开发人员，那么欢迎来到互联网通信领域！借助应用程序中的实时语音和视频功能，你可以让用户很好地与对方互动。要理解我们的 API 描述，你需要有 HTML、JavaScript 的相关知识和一些网页应用程序的经验。可以参阅附录 D 获取一些有用的参考。

如果你是一名 VoIP 或者电话开发人员，那么欢迎来到网络世界！你的用户将享受到高质量的音频和视频通信功能及以网络驱动的丰富的用户界面。为了能够更好地理解我们对用于传输语音、视频和数据的线上协议的描述，你需要对互联网有一个基本的了解。而掌握如 SIP 或 Jingle 之类的另一种互联网通信信号协议的知识也是非常有用的。可以参阅附录 D 获取一些有用的参考。

在许多方面，WebRTC 是网络世界和电话世界之间的一种融合。为了帮助弥合网络世界和电话世界之间的鸿沟，附录 C 中包含一个术语表来简要解释每个领域的一些常见术语和概念。

笔者期待着参与 WebRTC 可能会发起的下一个毁坏和革新浪潮。我们希望收到你的来信，并和我们在推特（@alanbjohnston 和 @danielcburnett）或者 Google+(alanbjohnston@gmail.com 和 danielcburnett@gmail.com) 上进行互动。

作者简介 *About the Author*

Alan B. Johnston 博士拥有超过 13 年的 SIP、IP 语音（Voice over IP，VoIP）和互联网通信经验，参与编著了 SIP 规范和许多其他 IETF RFC，其中包括 ZRTP 媒体安全协议。他著有四本关于互联网通信、SIP 和安全的技术类畅销书，还有一本科技惊险小说《Counting from Zero》，其中传授了互联网和计算机安全的基础知识。他是 SIP 论坛的董事会成员，拥有电气工程专业的学士和博士学位。Alan 是 IETF RTCWEB 工作组的积极参与者。他目前是 Avaya 公司的杰出工程师和圣路易斯华盛顿大学的兼职讲师。他拥有很多摩托车，并酷爱骑行摩托，他还为一个机器人团队提供辅导，并乐享其中。

Daniel C. Burnett 博士拥有 10 多年的计算机标准工作经验，曾编写和编辑了 W3C 的许多标准，这些标准为当今的大多数自动化交互式语音应答（Interactive Voice Response，IVR）系统奠定了基础。由于在自动语音识别（声音辨别）领域的标准制定方面贡献卓越，他曾两度荣获由《Speech Tech》杂志颁发的久负盛名的“语音杰出人物”奖。作为 PeerConnection 和 getUserMedia W3C WEBRTC 规范的编辑以及 IETF 的参与者，Daniel 从一开始就投入到了这个令人振奋的新领域。他目前是 Tropo 的首席科学家和 Voxeo（Aspect 旗下的一家公司）的标准总监。闲暇时间，Daniel 喜欢和家人及儿子的童子军团一起露营。

关注 Alan 和 Daniel 的推特账号 @alanbjohnston 和 @danielcburnett，以及他们的 Google+ 账号 alanbjohnston@gmail.com 和 danielcburnett@gmail.com。

有关未来版本以及发布后的更新和变更的信息，请访问 http://webrtcbook.com。

Facebook：http://www.facebook.com/webrtcbook

Google+：http://plus.google.com/102459027898040609362

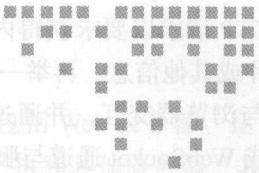
Acknowledgements 致　谢

我们要感谢技术审校者 Alex Agranovsky、Carol Davids、Emil Ivov、David Kemp、Henry Sinnreich、Harvey Waxman 和 Dan York。我们也要感谢 Marina Burnett 和 Chris Comfort 的校对和审校。同时我们也要感谢家人的鼓励和支持。

最后，我们要感谢在万维网联盟（W3C）的同事们和正在为制定 WebRTC 标准而不懈努力的国际互联网工程任务组（IETF）。

感谢声网（Agora.io）全体员工，特别是陈功、李伟和王骅的校对及评论。

目　录 *Contents*



第 1 章 Chapter 1

Web 实时通信技术介绍

Web 实时通信技术（Web Real-Time Communication，WebRTC）为 Web 浏览器增加了新的功能。它开创性地使得浏览器能够直接与其他浏览器交互，从而形成包括三角模式和梯形模式在内的多种体系结构。WebRTC 的多媒体功能非常先进，能提供很多新的特性。目前由万维网联盟（World Wide WebConsortium，W3C）和互联网工程任务组（Internet Engineering Task Force，IETF）联合负责 WebRTC 的标准化工作。

1.1 WebRTC 介绍

WebRTC 是一个基于标准化技术的行业性项目，旨在将实时通信功能引入到所有浏览器中，并通过标准的 [HTML5] 标签和 JavaScriptAPI 使这些功能可为 Web 开发者所用。打个比方，它提供类似于 Skype [SKYPE] 的功能，但不必安装任何软件或插件。为了让网站或 Web 应用顺畅地使用 WebRTC，需要得到所有的浏览器的支持，就必须为 WebRTC 制定相关的标准。同样地，为了让浏览器能够与非浏览器（包括企业和服务提供商的电话和其他通信设备）通信，同样需要为 WebRTC 制定相关的标准。

1.1.1 Web 浏览模式

Web 应用的基本模式如图 1.1 所示。通过 HTTP（超文本传输协议，详情请阅读 10.2.1 节）在浏览器与 Web 服务器之间传输数据。HTTP 运行于 TCP（传输控制协议，详情请阅读 10.2.9 节）的上层，或者，在某些新的网络实现中，HTTP 运行于 WebSocket 协议之上（详情请阅读 10.2.2 节）。在网络上使用 HTML（超文本标记语言）来承载内容或应用，典型

的 HTML 包括 JavaScript 和 CSS（级联样式表）。举一个简单的例子，浏览器向 Web 服务器发送一个 HTTP 请求以要求获得内容，然后，Web 服务器发送一个回应，其中包含被请求的文件或图片或其他信息。再举一个复杂的例子，服务器发送 JavaScript 运行在浏览器之上，通过 API 与浏览器交互，并通过用户的点击和选择操作与用户交互。浏览器通过一个开放式 HTTP 或 WebSocket 通道与服务器交换信息。

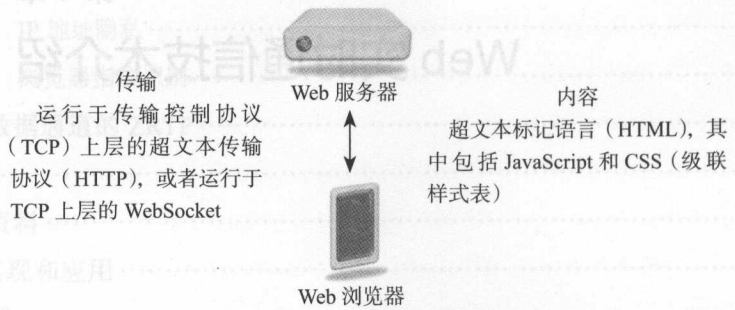


图 1.1 Web 浏览器模式

图 1.1 在 Web 浏览器和 Web 服务器之间显示箭头，以表明两者之间的 Web 会话。考虑到 WebRTC 能利用任何 Web 传输，本书将不讨论连接细节以及连接时使用 HTTP 还是 WebSocket。

1.1.2 浏览器中的实时通信功能

图 1.2 显示了浏览器模式，以及实时通信功能在其中发挥的作用。本书重点放在“浏览器实时通信功能”这一部分。实时通信技术的独特性质和要求，决定了增加该功能和实现该功能的标准化绝非易事。实时通信功能通过标准 API 与 Web 应用程序交互，并使用浏览器与操作系统通信。WebRTC 增加了一个新特点，即浏览器与浏览器之间的交互（所谓“对等连接”）。在此类交互中，一个浏览器之中的实时通信功能使用线上（on the wire）标准协议（非 HTTP），与另一个浏览器或网络电话（VoIP）或视频应用程序之中的实时通信功能进行通信。虽然使用 TCP 传输 Web 流量，但浏览器之间的线上协议可使用其他传输协议，例如 UDP（User Datagram Protocol，用户数据报协议）。WebRTC 的另一个新特点是提供信令服务器，该服务器在浏览器和对等连接另一端

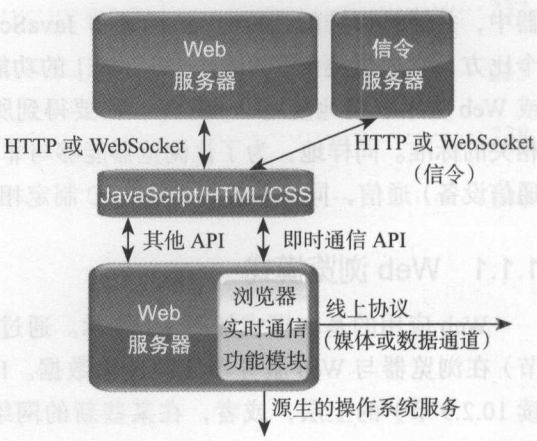


图 1.2 浏览器之中的实时通信功能

之间提供信令通道。

1.1.3　WebRTC 系统所含的元素

图 1.3 展现了 WebRTC 系统所含的典型元素集，包括 Web 服务器、运行于各种设备和操作系统之上的浏览器，包括台式机、平板电脑、手机和其他服务器。其他元素还包括 PSTN（公用交换电话网）门户以及其他互联网通信终端，例如 SIP（会话发起协议）电话、客户端，或者 Jingle 客户端。WebRTC 支持上述所有设备之间的通信。本书中的各图将使用这些标识和元素进行举例。

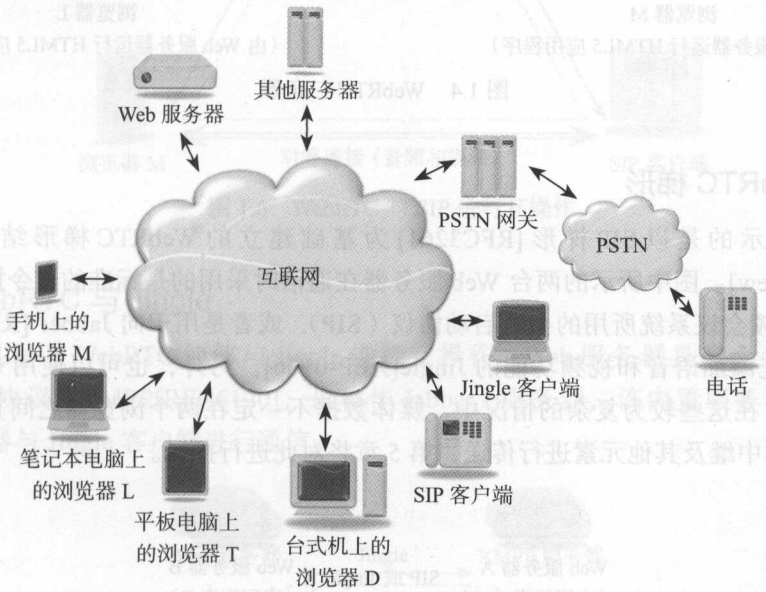


图 1.3　WebRTC 系统所含的元素

1.1.4　WebRTC 三角形

起初，最常见的情形很可能是两个浏览器都在运行从同一个 Web 服务器下载的同一个 WebRTC Web 应用程序。这就会形成图 1.4 中所示的 WebRTC“三角形”。这种布局之所以称为三角形，是因为这三个元素之间的信令路线（三角形的两条侧边）和媒体或数据流动路线（三角形的底边）所构成的形状恰似一个三角形。两个浏览器之间直接通过建立对等连接来传输语音、视频媒体，以及附加数据。

请注意，虽然我们有时会将浏览器与服务器之间的连接称作信令，但它其实并非指电话系统中所用的信令。信令在 WebRTC 中并未实现标准化，因为它只是被视作应用程序的一部分。信令可以通过 HTTP 或 WebSocket 传送到向浏览器提供 HTML 页面的同一 Web 服务器，也可传送到只负责处理信令的一个完全不同的 Web 服务器。

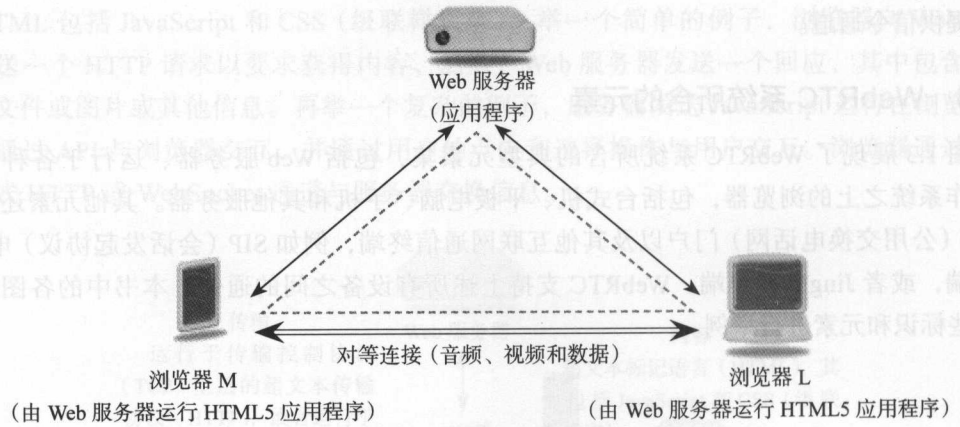


图 1.4 WebRTC 三角形

1.1.5 WebRTC 梯形

图 1.5 显示的是以 SIP 梯形 [RFC3261] 为基础建立的 WebRTC 梯形结构 [draft-ietf-rtcweb-overview]。图中所示的两台 Web 服务器在通信时采用的是标准的信令协议，例如很多 VoIP 和视频会议系统所用的会话启动协议（SIP），或者是用于向 Jabber [RFC6120] 即时消息现场系统添加语音和视频功能的 Jingle[XEP-0166]。另外，也可以使用专有的信令协议。请注意，在这些较为复杂的情况中，媒体数据不一定在两个浏览器之间直接传输，而是可通过媒体中继及其他元素进行传送，第 5 章将对此进行探讨。

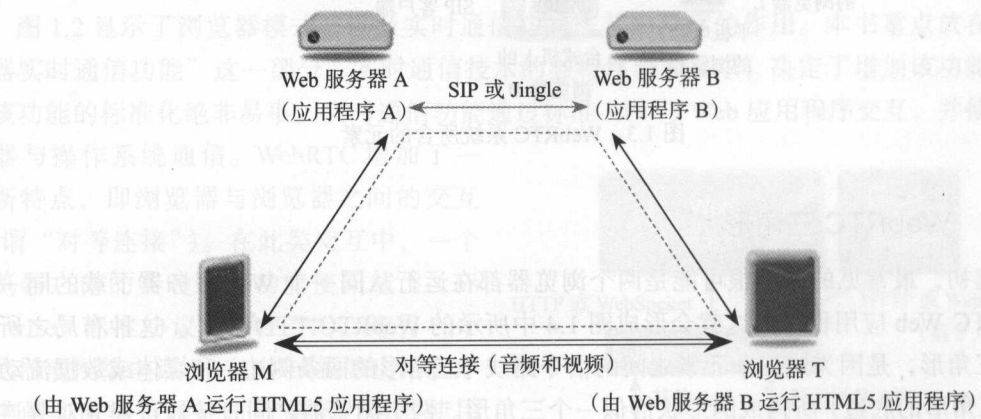


图 1.5 WebRTC 梯形

1.1.6 WebRTC 和会话启动协议 SIP

图 1.6 显示的是与会话启动协议（SIP）进行互操作的 WebRTC。Web 服务器有一个内

置的 SIP 信令网关，浏览器与 SIP 客户端之间可通过此网关来交换呼叫建立信息。这会直接在浏览器与 SIP 客户端之间形成媒体流，因为对等连接会与 SIP 用户代理建立标准的实时传输协议（RTP）媒体会话（参见 10.2.3 节）。2.2.6 节介绍与 SIP 进行互操作的其他一些方式。

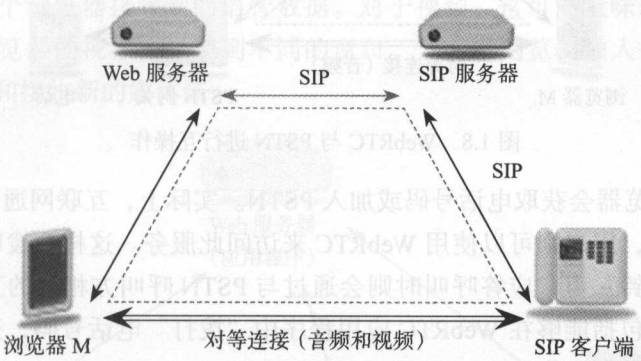


图 1.6 WebRTC 与 SIP 进行互操作

1.1.7 WebRTC 与 Jingle

图 1.7 显示了 WebRTC 如何与 Jingle 进行互操作。Web 服务器具有一个内置的可扩展消息现场协议（XMPP[RFC6120]，也称作 Jabber）服务器，该内置服务器通过另一个 XMPP 服务器与 Jingle 客户端进行通信。

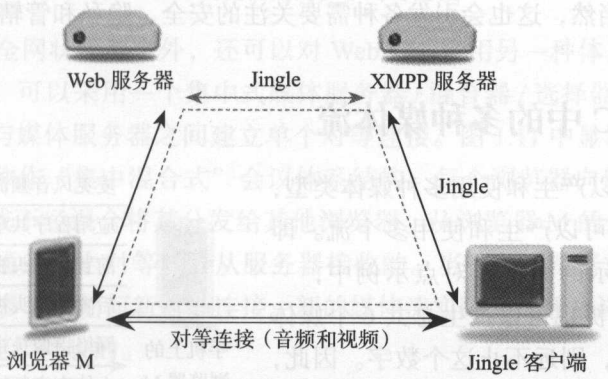


图 1.7 WebRTC 与 Jingle 进行互操作

1.1.8 WebRTC 与公共交换电话网

图 1.8 显示了 WebRTC 如何与公共交换电话网（PSTN）进行互操作。PSTN 网关是纯音频媒体流的终结点，负责将 PSTN 电话呼叫与媒体相连。在 Web 服务器与 PSTN 网关之间需要有某种形式的信令，具体可以是 SIP，也可以是主 / 从控制协议。

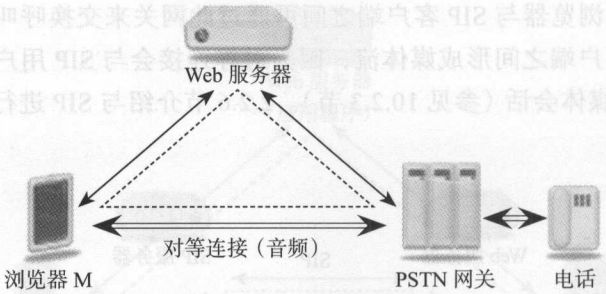


图 1.8 WebRTC 与 PSTN 进行互操作

并没有预期浏览器会获取电话号码或加入 PSTN。实际上，互联网通信服务可以为用户分配一个电话号码，用户则可以使用 WebRTC 来访问此服务。这样，拨叫该电话号码时浏览器就会发出“响铃”声，应答呼叫时则会通过与 PSTN 呼叫方相连的互联网建立语音会话。其他服务可能包括能够在 WebRTC 应用程序中“拨打”电话号码，这种拨叫会通过互联网与 PSTN 建立语音通道。

请注意，图 1.8 中的电话可以是普通的 PSTN 电话（“座机”或“固定电话”），也可以是移动电话。如果运行的是 VoLTE 或其他 VoIP 协议，情况也不例外，因为对等连接将终结于 VoIP 网关。

另一个值得关注的方面是 WebRTC 在提供紧急服务过程中所发挥的作用。虽然 WebRTC 服务可以采取与 VoIP 互联网通信服务相同的方式来支持紧急呼叫，但公共服务应答点（PSAP）可能会成为 WebRTC 应用程序，并直接从其他浏览器应答紧急“呼叫”，从而完全绕过 PSTN。当然，这也会引发各种需要关注的安全、隐私和管辖权问题。

1.2 WebRTC 中的多种媒体流

如今的设备可以产生和使用多种媒体类型，对于每种类型，则可以产生和使用多个流。即使是在图 1.9 中所示的简单点对点示例中，一部手机和一台台式机也可以产生总共 6 个媒体流。对于多方会话，则远不止这个数字。因此，WebRTC 内置了针对多媒体流和多数据来源的处理功能。

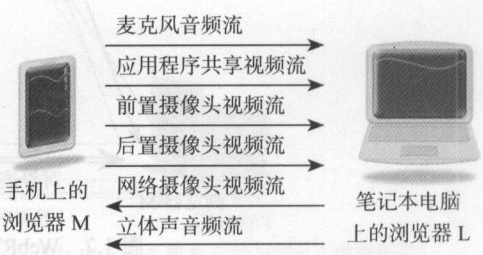


图 1.9 点对点 WebRTC 会话中的多媒体流

1.3 WebRTC 中的多方会话

前面的几个示例一直是两个浏览器之间或一个浏览器与另一个终端之间的点对点会话。

WebRTC 还支持有多个浏览器参与的多方会话或会议会话。

要建立这类会话，一种方式就是让每个浏览器与参与会话的其他浏览器建立一个对等连接。图 1.10 中显示了具体的连接结构。这种结构有时也称作“全网状”或“完全分布式”会议体系结构。每个浏览器均与其他浏览器建立全网状的对等连接。对于音频，这可能意味着需要混合从每个浏览器接收到的语音数据。对于视频，这可能意味着需要使用相应的标记将来自其他浏览器的视频流呈现到不同的窗口。当有新浏览器加入会话时，会建立新的对等连接来发送和接收新的媒体流。

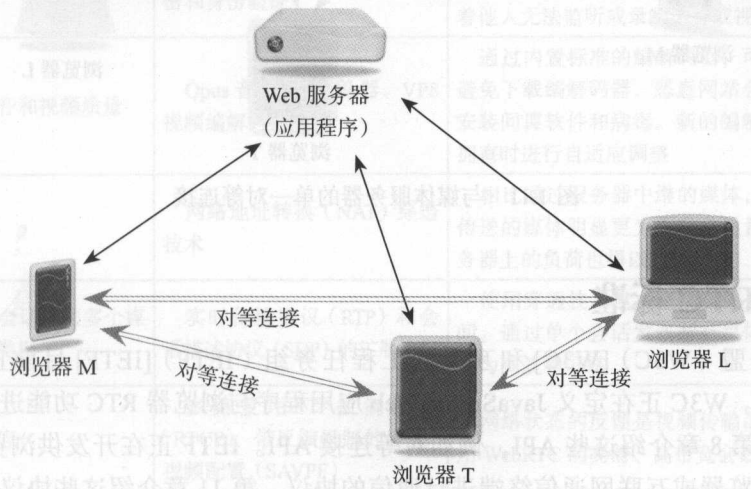


图 1.10 浏览器间的多个对等连接

除了图 1.10 中的全网状模型之外，还可以对 WebRTC 采用另一种体系结构。对于有多个浏览器参与的会议，可以采用一个集中式媒体服务器 / 混合器 / 选择器；在这种情况下，只需要在每个浏览器与媒体服务器之间建立单个对等连接。图 1.11 中显示了具体的连接结构。这种结构有时也称作“集中混合式”会议体系结构。每个浏览器向服务器发送媒体数据，由服务器混合后或不经混合将其分发给其他浏览器。从浏览器 M 的角度看，来自浏览器 L 和 T 的媒体流是通过单个对等连接从服务器接收的。当有新浏览器加入会话时，不需要重新建立浏览器 M 参与的任何新对等连接。新的媒体流实际上是通过浏览器 M 与媒体服务器之间的现有对等连接接收的。

图 1.10 的全网状体系结构具有的优势是，不需要媒体服务器基础架构、媒体延迟最低且质量最高。不过，这种体系结构可能不适合大型的多方会议，因为随着新参与者的加入，每个浏览器所需的带宽也随之增加。图 1.11 中的集中式体系结构具有如下优势：能够扩展为非常大的会话，同时还可以最大限度减少当有新参与者加入会话时每个浏览器所需的处理工作量。不过，当只有一个浏览器或少量浏览器参与时（例如点对点游戏），这种体系结构的效率可能较低。

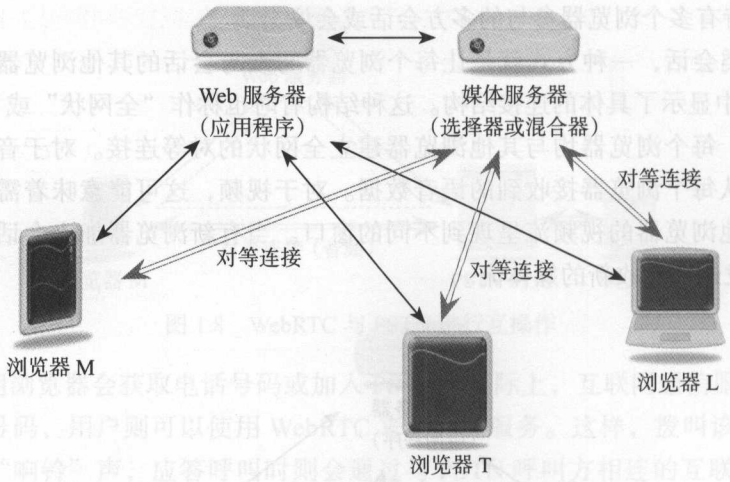


图 1.11　与媒体服务器的单一对等连接

1.4　WebRTC 标准

万维网联盟（W3C）[W3C] 和互联网工程任务组（IETF）[IETF] 目前正在联合制定 WebRTC 标准。W3C 正在定义 JavaScript Web 应用程序与浏览器 RTC 功能进行交互时需要用到的 API。第 8 章介绍这些 API，例如对等连接 API。IETF 正在开发供浏览器 RTC 功能用来与其他浏览器或互联网通信终端进行通信的协议。第 11 章介绍这些协议，例如实时传输协议的各种扩展协议。

目前一些浏览器中采用的很多 WebRTC 组件都是在标准尚未正式出台时先行实现的。有关详情，请参见第 14 章。

请注意，“标准出台前”的实现方案与“专有”实现方案之间存在重要区别。标准出台前的实现方案是在标准制定阶段形成的，对于获得相应的经验和信息来敲定和锁定标准至关重要。标准出台前的实现方案通常遵照标准的早期版本或草拟版本，或者在一定程度上执行标准以作为一种“概念验证”。标准敲定下来后，这些在标准出台前形成的实现方案必须向标准靠拢，否则它们将有可能成为专有实现方案。专有实现方案不能形成统一的用户和开发基础，在通信等领域这可能会大大降低服务的价值。

W3C 的工作以 WEBRTC 工作组为中心开展，IETF 的工作则围绕着 RTCWEB（实时通信 Web）工作组开展。这两个工作组相互独立，但彼此之间密切配合，并且有很多共同的参与者，包括撰稿人。

1.5　WebRTC 的新功能

WebRTC 新增了很多激动人心的功能，即便是目前的 VoIP 和视频会议系统也尚未提供

这些功能。表 1.1 列出了其中一些功能。本书其余部分将讲解这些功能是如何使用 WebRTC API 和协议实现的。

表 1.1 WebRTC 的新功能特性

功能 / 特性	使用什么实现	为何重要
平台和设备无关性	W3C 提供的标准 API、IETF 提供的标准协议	开发人员可以编写跨操作系统、浏览器和设备（台式机以及手机）运行的 WebRTC HTML5 代码
安全语音和视频	安全 RTP 协议（SRTP）加密和身份验证	浏览器可以在各种不同的环境中使用，并且可以通过未受保护的 WiFi 网络加以使用。加密意味着他人无法监听或录制语音或视频
非同一般的语音和视频质量	Opus 音频编解码器、VP8 视频编解码器等	通过内置标准的编解码器，可确保互操作性并避免下载编解码器，恶意网站会通过这种下载来安装间谍软件和病毒。新的编解码器可在检测到拥塞时进行自适应调整
建立可靠会话	网络地址转换（NAT）穿透技术	相比通过服务器中继的媒体，直接在浏览器间传送的媒体明显更为可靠且质量更高。此外，服务器上的负荷也得以减轻
通过单个传输会话发送多个媒体流和多种媒体类型	实时传输协议（RTP）和会话描述协议（SDP）的扩展协议	使用穿透技术建立直接媒体连接需要一定的时间。通过单个会话发送所有媒体不仅省时，而且更为高效和可靠
网络状态自适应	多路复用 RTP 控制协议（RTCP）、带反馈机制的音频 / 视频配置（SAVPF）	网络状态的反馈是视频传输的关键，而对于采用 WebRTC 的高清、高带宽会话来说则尤为重要
支持多种媒体类型和多种媒体来源	用于逐一协商每种来源的大小 / 格式的 API 及信令	由于能够逐一协商每种来源，因此带宽及其他资源可以得到最高效的利用
能够使用 SIP、Jingle 和 PSTN 与 VoIP 和视频通信系统进行互操作	标准安全 RTP（SRTP）媒体、标准 SDP 和扩展协议	现有的 VoIP 和视频系统可以使用标准协议与新的 WebRTC 系统协同发挥作用

1.6 重要的术语说明

在本书中，我们用 WebRTC 来指代向浏览器中添加标准化通信功能的全部工作，用 WEBRTC 来指代 W3C 工作组，用 RTCWEB 来指代 IETF 工作组。请注意，WebRTC 还用于描述 Google/Mozilla 开源媒体引擎 [WEBRTC.ORG]，这种引擎也是 WebRTC 的一种实现形式。

此外，由于主要 W3C 规范的名称为“WebRTC 规范”[WEBRTC 1.0]，因此我们使用该规范的全名来引用此特定 W3C 文档，此文档是 WebRTC 的关键组成部分，但绝非指整个规范。

还请注意，万维网联盟将自身称作“W3C”，而非“theW3C”。在本书中我们已采用这一惯例。

1.7 参考资料

[HTML5] http://www.w3.org/TR/html5

[SKYPE] http://www.skype.com

[CSS] http://www.w3.org/Style/CSS

[draft-ietf-rtcweb-overview] http://tools.ietf.org/html/draft-ietf-rtcweb-overview

[RFC3261] http://tools.ietf.org/html/rfc3261

[XEP-0166] http://xmpp.org/extensions/xep-0166.html

[RFC6120] http://tools.ietf.org/html/rfc6120

[W3C] http://www.w3c.org

[IETF] http://www.ietf.org

[WEBRTC.ORG] http://www.webrtc.org

[WEBRTC 1.0] http://www.w3.org/TR/webrtc

第 2 章 Chapter 2

如何使用 WebRTC

WebRTC 易于使用，只需少数几个步骤即可建立媒体会话。有些消息在浏览器和服务器之间流动，有些则直接在两个浏览器（称为对等端）之间流动。WebRTC 甚至支持与 SIP、Jingle 和 PSTN 终端建立会话。WebRTC 技术涉及大量的标准，由于数量众多，初学者很难判断该从何处着手。由于本书的许多读者可能会从事 WebRTC 应用程序开发工作，因此以下章节将大体介绍如何建立 WebRTC 会话、会话运行期间可执行哪些操作，以及如何关闭会话等。WebRTC 适用于许多不同的架构。伪代码示例说明了 WebRTC 应用程序编程接口（Application Programming Interface，API）的工作方式。后续章节将详细介绍其中的每个关键环节。

2.1 建立 WebRTC 会话

作为应用程序开发者，建立 WebRTC 会话时需要以下四个主要步骤：

1）获取本地媒体。

2）在浏览器和对等端（其他浏览器或终端）之间建立连接。

3）将媒体和数据通道关联至该连接。

4）交换会话描述。

图 2.1 显示了这四个步骤。请注意，此图与图 1.4 相对应。

图 2.2 显示了另一个步骤视图，其中增加了信令建立步骤。

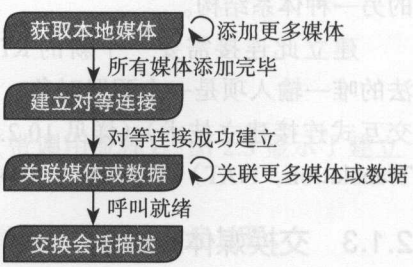


图 2.1 建立 WebRTC 会话（API 视图）

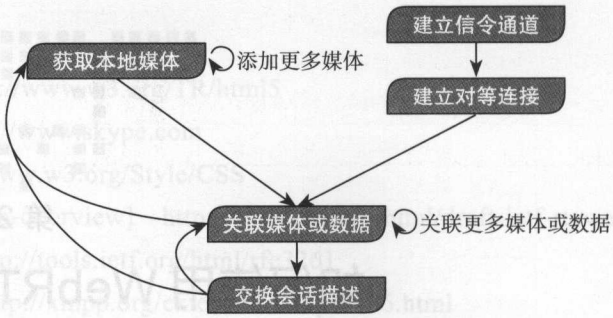


图 2.2 WebRTC API 视图（含信令）

以下几个小节将简要介绍其中的每个步骤，以及通信结束时的会话关闭过程。

2.1.1 获取本地媒体

有多种方式可用于获取媒体，关于完整的方式清单，本书不再赘述。但是，WebRTC 技术定义了一种最常见的方式：getUserMedia()（详见 8.3.2 节）。此方法可用于获取单个本地 MediaStream。在获取一个或多个媒体流后，可使用 MediaStream API 将它们组合到所需的流中。为保护隐私，只有当浏览器获得用户的许可后，才会批准 Web 应用程序访问用户麦克风或摄像头的请求。

2.1.2 建立对等连接

另一个重要步骤是使用相应的 API 建立对等连接。RTCPeerConnection API 是 WebRTC 的核心，顾名思义，其作用就是在两个对等端之间建立连接。在本书中，“对等端”是指万维网上的两个通信终端，与术语“对等端文件共享”中的对等端同义。这不是通过服务器请求通信，而是直接在两个实体之间进行通信。就 WebRTC 本身而言，对等连接就是两个 Web 浏览器之间的直接媒体连接。这种模式非常实用，尤其适用于多向通信，例如在三个或更多浏览器之间建立的电话会议。每一对浏览器都需要一个对等连接才能加入会议，这样音频和视频流可直接在两个浏览器之间流动，如图 1.10 所示。因此，如果有三个浏览器进行通信，总共将需要建立三个连接。应用程序开发人员将需要为每一对要连接的浏览器（或浏览器与其他终端，例如现有通信网络）建立一个对等连接。也可以采用图 1.11 中所示的另一种体系结构。

建立此连接需要一个新的 RTCPeerConnection 对象。RTCPeerConnection 构造函数方法的唯一输入项是一个配置对象，该对象包含 ICE（Interactive Connectivity Establishment，交互式连接建立技术）（详见 10.2.7 节）“打洞”通过各种网络地址转换（Network Address Translation，NAT）设备和防火墙时所使用的信息。

2.1.3 交换媒体或数据

建立连接后，可将任意数量的本地媒体流关联到对等连接，以通过该连接发送至远端

浏览器。同样，也可以将任意数量的远端媒体流发送至对等连接的本地端，这样本地端就有了新的媒体流，而且可以像操作其他本地媒体流那样处理它们。

需要指出的是，每次更改媒体时，都需要在两个浏览器之间协商（或重新协商）如何在连接通道中表示媒体。当从本地端或远程端发出添加或删除媒体的请求时，可请求浏览器生成相应的 RTCSessionDescription 对象（此容器存放会话描述，即有关如何建立媒体会话的信息），用于表示通过对等连接传输的所有媒体集合。利用 RTCPeerConnection API，应用程序开发者可在会话描述被发送至远端之前，根据需要查看和编辑会话描述。在这种设计下，浏览器既可处理编解码器协商和编写会话描述协议（Session Description Protocol，SDP）（用于表示会话描述，详见 10.2.4 节）此类的“繁重工作”，同时仍允许应用程序根据需要进行微调。然而，在大多数情况下，Web 开发人员应该不需要修改或检查 RTCSessionDescription 对象。

当两个浏览器交换完 RTCSessionDescription 对象后，即可建立媒体或数据会话。此时，两个浏览器将开始打洞。打洞完毕后，即可开始协商密钥，以确保媒体会话的安全。最后可开始媒体或数据会话。请注意，交换 RTCSessionDescription 对象后的所有这一切活动通过浏览器执行 JavaScript 代码完成。应用程序的 JavaScript 代码还可以添加 / 删除 STUN 和 TURN 服务器（用于 NAT 穿透和监控这一过程，由浏览器完成，详见 10.2.5 节和 10.2.6 节）。

2.1.4 关闭连接

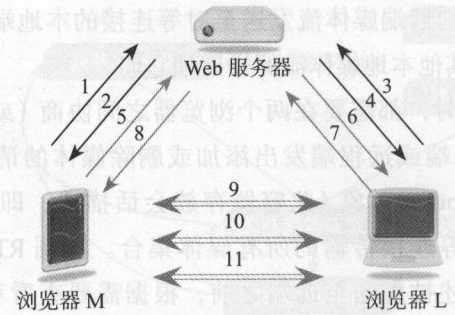
在对等连接中，任何一端的浏览器都可以关闭连接。应用程序通过对 RTCPeer Connection 对象调用 close() 来指示连接已使用完毕，这可能是对用户单击按钮或关闭标签页做出的响应。此操作会停止 ICE 处理和媒体流传输。同样，如果某一端的浏览器断开 Internet 连接或发生崩溃，媒体或数据通道中发送的持久连接（keep-alive）请求将失效，另一端的浏览器将尝试重新开始打洞，并在打洞失败时关闭会话。会话结束后，浏览器将删除该会话授予的任何访问设备麦克风和摄像头的许可，因此在新的会话中，需要向用户请求新的许可。

2.2 WebRTC 联网和交互示例

以下小节将展示一些 WebRTC 应用程序的体系结构（带调用流程）。图 2.3 显示了建立 WebRTC 会话的协议视图。

以下图形以“梯形图”（有时称为调用流程）的形式详细展示了协议交换过程。

诸如 SRTP、SDP 和 ICE 等协议将在第 10 章中介绍。



注意：只要 M 和 L 正在运行同一 Web 应用程序，则对于步骤 1 和 2 以及步骤 3 和 4 来说，孰先孰后并不重要。

1）浏览器 M 从 Web 服务器请求网页。
2）Web 服务器向 M 提供带有 WebRTC JavaScript 的网页。
3）浏览器 L 从 Web 服务器请求网页。
4）Web 服务器向 L 提供带有 WebRTC JavaScript 的网页。
5）M 决定与 L 通信，并通过 M 上的 JavaScript 将 M 的会话描述对象（offer，提议）发送至 Web 服务器。
6）Web 服务器将 M 的会话描述对象发送给 L 上的 JavaScript。
7）L 上的 JavaScript 将 L 的会话描述对象（answer，应答）发送至 Web 服务器。
8）Web 服务器将 L 的会话描述对象发送给 M 上的 JavaScript。
9）M 和 L 开始打洞，以确定访问对方的最佳方式。
10）完成打洞后，M 和 L 开始为安全的媒体通信协商密钥。
11）M 和 L 开始交换语音、视频或数据。

图 2.3　建立 WebRTC 会话（协议视图）

2.2.1　在 WebRTC 三角形中建立会话

图 2.4 显示了使用 WebRTC 建立基本会话的过程。请注意，图 2.4 与图 2.3 相对应。

浏览器 M 和 L 正在运行从 Web 服务器下载的同一个支持 WebRTC 的 JavaScript。当一位用户希望与另一位用户通信时，将开始在两个浏览器之间进行媒体协商，此过程称为提议 / 应答交换。此提议 / 应答交换通过两个浏览器之间建立的信令通道进行。在本章剩余部分中，假定该信令通道通过 HTTP 连接 Web 服务器。有关此信令通道的工作方式以及备选信令通道设计的详情，请参阅第 4 章。

媒体协商是指通信会话中的双方（例如两个浏览器）进行通信并就可接受的媒体会话达成一致的过程。“提议 / 应答”是一种媒体协商方式：首先，一方向另一方发送其支持并要设置的媒体类型和功能，这称为“提议”；随后，另一方予以回应，指示在所提议的媒体类型和功能中，哪些是此会话支持并可以接受的，这称为“应答”。为建立和修改会话，例如添加新的媒体流或更改要发送的媒体流，此过程可能会重复多次。人们经常会问：为何采取这样一个来回反复的过程，而不是让每一方只表明自己的意图？这主要是为了确保双方在媒体流传输之前达成一致。这一点非常重要，因为在浏览器准备就绪之前，浏览器中底层模块可能无法处理传入的媒体流。

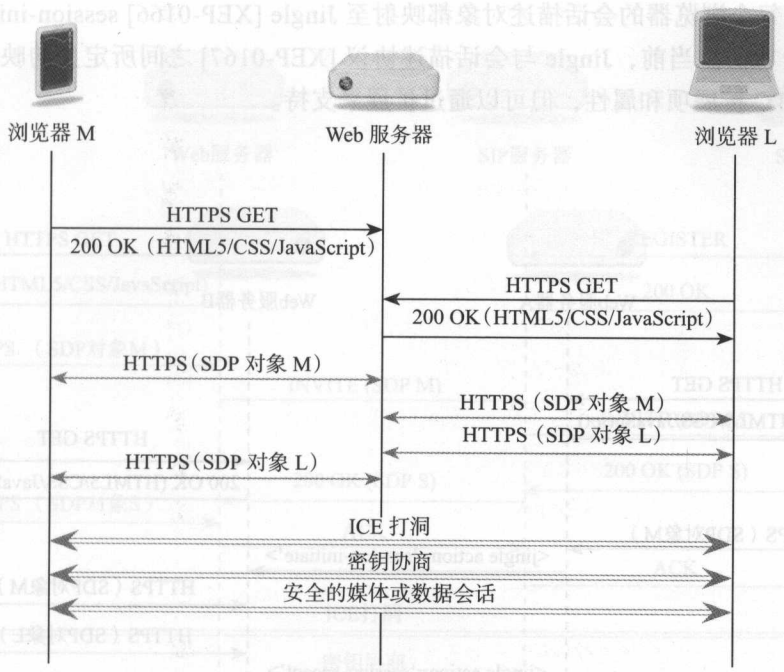


图 2.4　WebRTC 三角形调用流程

除了提议和应答外，还有一种“临时应答”（pranswer）。临时应答是对提议的应答，但是临时性或试验性的。它可能不是实际应答中提供的最终应答，最终应答将在后面的提议 / 应答交换中给出。临时应答是可选的，通常只在与某些模拟电话网络特征（early media，早期媒体）的 VoIP 系统或 PSTN 进行互操作时，才会出现。

当浏览器 M 的用户决定与浏览器 L 的用户通信时，浏览器 M 的 JavaScript 将针对其需要的媒体提供基于约束的描述、请求媒体数据并获取用户许可。用户授予的许可必须绑定到网页所在的域，并且不能扩展到网页上的弹出窗口和其他框架，这一点非常重要。所需的媒体会话信息由会话描述对象捕获。该对象就是通过 Web 服务器向浏览器 L 发送的提议（offer）。需要注意的是，对于浏览器 M 如何将此提议发送至浏览器 L，WebRTC 并未提供标准化方法。此过程可通过多种方式来完成，例如 XML HTTP 请求 [XHR]。收到会话描述对象提议后，浏览器 L 将生成会话描述对象应答，并以相同方式发送回浏览器 M。当提议 / 应答交换完成后，即可开始打洞（和密钥协商），并最终交换媒体数据包。

当浏览器 L 关闭连接时，将导致对等连接关闭，对麦克风和摄像头的许可也全部失效。

2.2.2　在 WebRTC 梯形中建立会话

图 2.5 显示了图 1.5 中所示的 WebRTC 梯形的调用流程。

此场景中，浏览器 M 和 L 直接交换媒体，只是它们运行的 Web 应用程序来自不同的

Web 服务器。每个浏览器的会话描述对象都映射至 Jingle [XEP-0166] session-initiate 消息和 session-accept 方法。当前，Jingle 与会话描述协议 [XEP-0167] 之间所定义的映射并不包括 SDP 的 WebRTC 扩展项和属性，但可以通过扩展来支持。

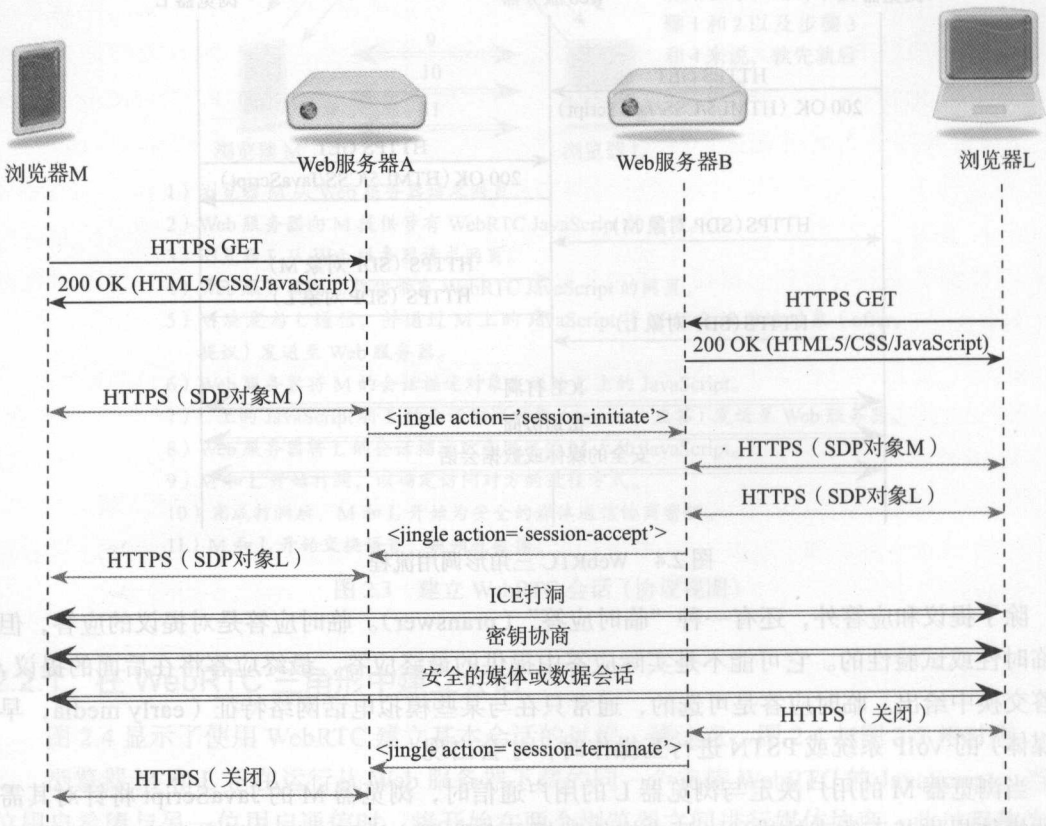


图 2.5　WebRTC 梯形调用流程（采用 Jingle）

请注意，在上述场景中，也可以将会话描述对象映射至 SDP，并在两个浏览器之间使用 SIP 信令。这无需更改 SIP 协议即可实现。

2.2.3　与 SIP 终端建立 WebRTC 会话

图 2.6 详细展示了 WebRTC 与 SIP [RFC3261] 的交互过程。此图与图 1.4 相对应。WebRTC 定义了会话描述对象提议 / 应答到 SIP 的映射，无需修改或扩展即可由常规 SIP INVITE 和 200 OK 消息传递。

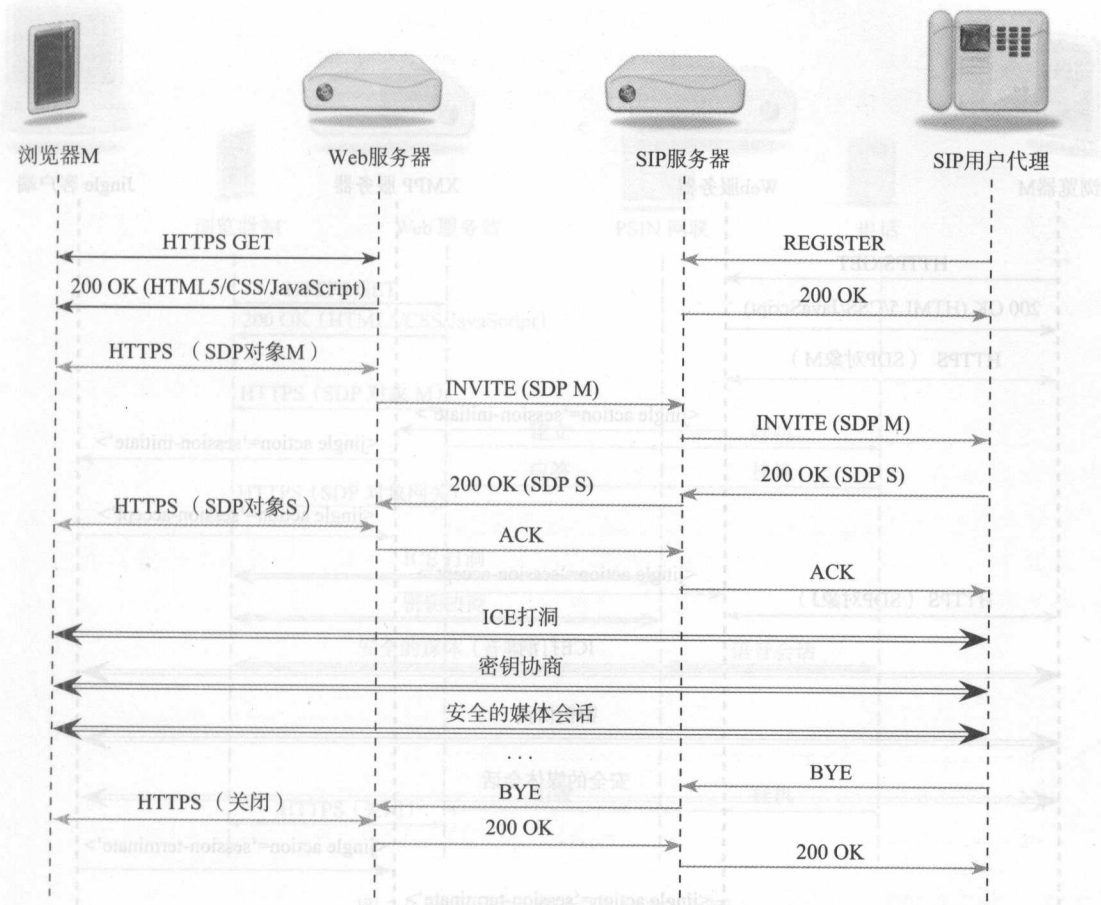


图 2.6　WebRTC 与 SIP 交互调用流程

2.2.4　与 Jingle 终端建立 WebRTC 会话

图 2.7 显示了 WebRTC 如何与 Jingle 进行互操作。图 2.7 与图 1.4 相对应，其展示的工作方式类似于 2.2.2 节中介绍的映射。媒体流可直接在浏览器和 Jingle 客户端之间传输，前提是扩展 Jingle 协议，使之支持 WebRTC SDP 扩展项，并且让 Jingle 客户端支持 WebRTC 媒体扩展项。

2.2.5　与 PSTN 建立 WebRTC 会话

图 2.8 显示了 WebRTC 如何与 PSTN 互操作。图 2.8 与图 1.7 相对应。Web 服务器和 PSTN 网关之间的信令可利用任意数量的信令或控制协议，甚至是 SIP 中继 [SIP-CONNECT]。PSTN 网关终止 WebRTC 媒体会话并将音频连接至 PSTN 中继或线路。应在浏览器和 PSTN 网关之间协商 G.711（PCM）编码格式，否则将需要对音频信号进行转码。

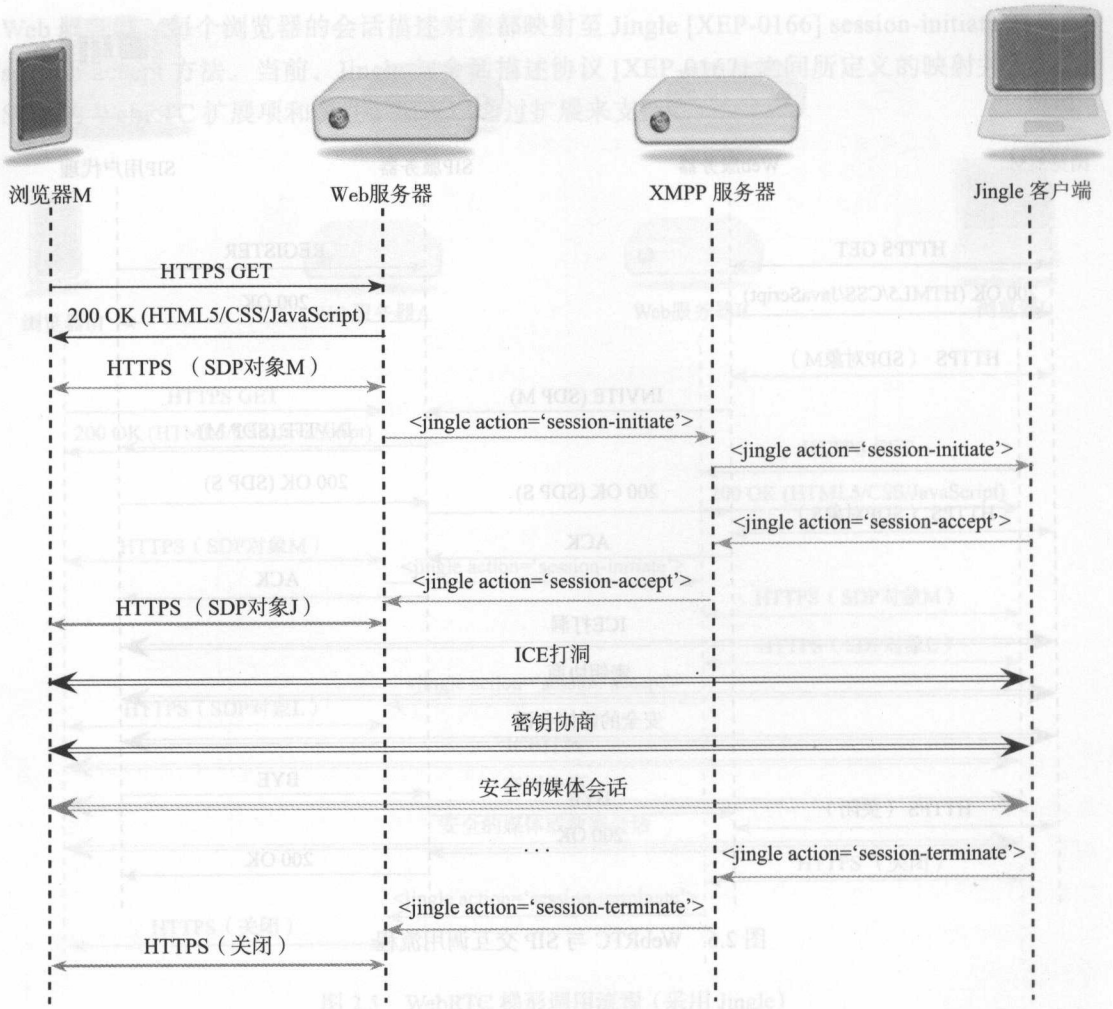


图 2.7 WebRTC 与 Jingle 互操作

2.2.6 与 SIP 和媒体网关建立 WebRTC 会话

图 2.9 和图 2.10 显示了 WebRTC 与 SIP 交互的另一种略微不同的方式。在此示例中，不再采取完全的端到端媒体传输，而是使用媒体网关来终止 ICE 和 SRTP，并且将媒体转发给 SIP UA，转发时甚至可能采取未加密的 RTP 形式；不过，只有在 VoIP 网络利用某种其他安全协议（例如 IPsec [IPSEC]）时才能如此。这是并非支持 WebRTC 所有扩展项的 VoIP 和视频终端与 WebRTC 交互的方式。

媒体网关也可以是一个边界元素，称为会话边界控制器（Session Border Controller，SBC），用于穿透企业或服务提供商网络的防火墙。

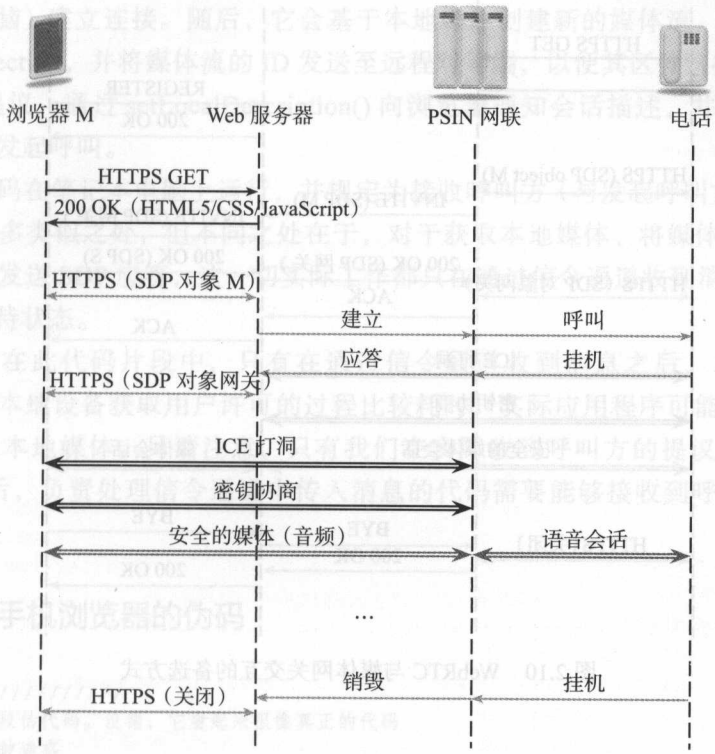


图 2.8　与 PSTN 建立 WebRTC 会话

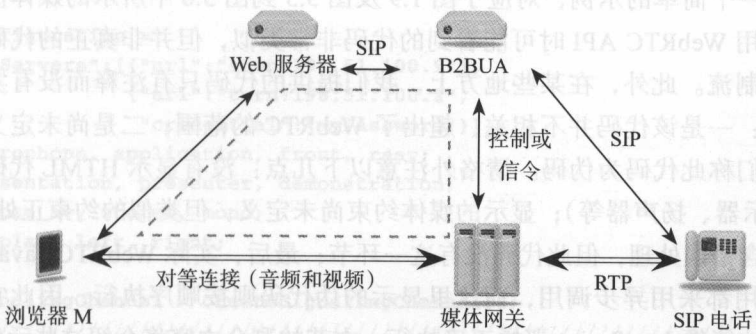


图 2.9　WebRTC 与 SIP 交互的备选方式

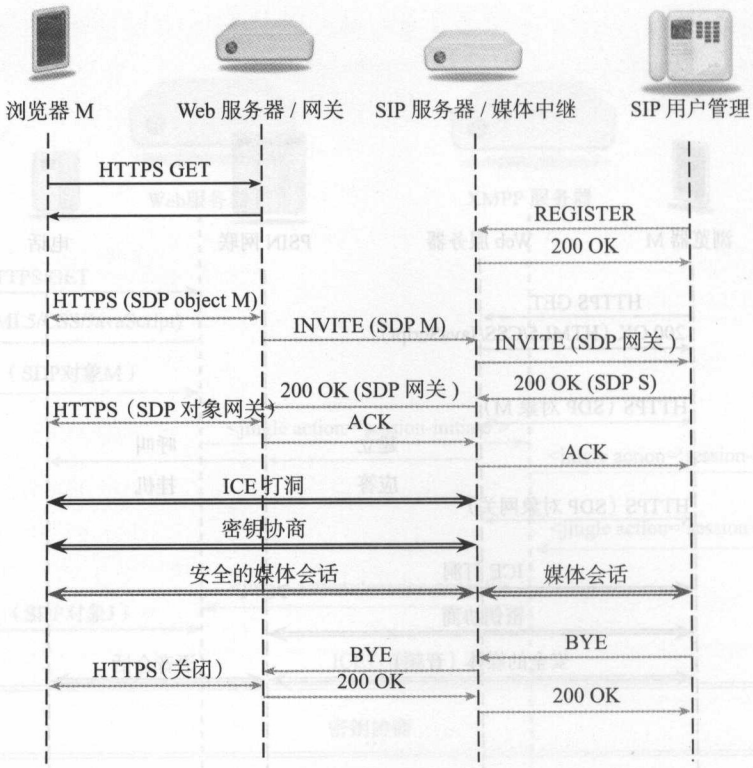


图 2.10　WebRTC 与媒体网关交互的备选方式

2.3　WebRTC 伪码示例

本节包含一个简单的示例，对应于图 1.9 及图 5.3 到图 5.6 中所示的媒体流。此示例代码与大家在使用 WebRTC API 时可能看到的代码非常类似，但并非真正的代码，其目的是让读者看清控制流。此外，在某些地方上，我们提供的代码只有注释而没有实际代码，这源于以下原因：一是该代码并不相关，超出了 WebRTC 的范围；二是尚未定义该代码。正因为如此，我们称此代码为伪码。请格外注意以下几点：没有显示 HTML 代码，但假定已设置输出（显示器、扬声器等）；显示的媒体约束尚未定义，但类似的约束正处在讨论之中；实际代码将包含错误处理，但此代码没有这一环节；最后，实际 WebRTC JavaScript 代码中的许多函数调用都采用异步调用，而这里显示的伪代码则按顺序执行，因此它必须等待每个回调完成，才能像 JavaScript 那样正确执行。本节的剩余内容将介绍这些示例，请在阅读时注意上述限制。（请注意，在本书接下来的几个章节中，我们将构建一个没有这些限制且可运行的真实代码示例。）

WebRTC 规范的第 10 节 [示例] 包含一个示例，其中展示了用于同时在呼叫方站

点和被叫方站点运行的代码。为便于理解，以下示例展示了两个独立的代码段，分别用于显示两个方向的不同媒体流，以及获取媒体的不同时间。第一段代码在移动设备上运行，并规定为发起呼叫方（与接收呼叫方相对）。它先获取本地媒体，再与远程对等端（笔记本电脑）建立连接。随后，它会基于本地媒体创建新的媒体流，将其关联到新的RTCPeerConnection，并将媒体流的ID发送至远程对等端，以便其区分这些媒体流。最后，它创建SDP提议，通过setLocalDescription()向浏览器通知会话描述，并将其发送至远程对等端，从而发起呼叫。

第二段代码在笔记本电脑上运行，并规定为接收呼叫方（与发起呼叫方相对）。它与第一段代码有很多类似之处，但不同之处在于，对于获取本地媒体、将媒体流关联到对等连接以及创建和发送SDP应答，这一切实际工作都只在通过信令通道收到消息后发生。此前则一直处于等待状态。

请注意，在此代码片段中，只有在通过信令通道收到消息之后，才会获取本地媒体。如果访问本地设备获取用户许可的过程比较耗时，实际应用程序可能有必要在等待呼叫之前就获取本地媒体。另请注意，只有我们在实际收到呼叫方的提议之后，才会调用answer()。最后，负责处理信令通道中传入消息的代码需要能够接收到呼叫方发来的媒体流ID。

2.3.1 针对手机浏览器的伪码

```
//////////////////
//  这是一段伪代码。没错，它看起来很像真正的代码
//  请不要被迷惑
//  不要认为它能够在某处运行！
//  本书接下来的几个章节将提供实际代码
//////////////////
var pc;
var configuration =
  {"iceServers":[{"url":"stun:198.51.100.9"},
                 {"url":"turn:198.51.100.2",
                  "credential":"myPassword"}]};
var microphone, application, front, rear;
var presentation, presenter, demonstration;
var remote_av, stereo, mono;
var display, left, right;

var signalingChannel = createSignalingChannel();
/////////////////////////////////////////////////////////////////
//  步骤0用于建立信令通道。关于如何实现这一点，没有限制要求
//  对等端的身份在建立信令通道期间确定
//  因此，RTCPeerConnection本身并不包含任何指示对等端身份的配置信息
//  下面的初始代码片段假定此信令通道具有send()方法和onmessage处理程序
//  前者将其参数发送至对等端，
```

```
//  并在对等端触发 onmessage 处理程序执行
////////////////////////////////////////////////////////////////////

////////////////////////////////////
// 以下是 4 个主要的函数调用
////////////////////////////////////

// 首先，获取本地媒体
getMedia();

// 然后，创建对等连接
createPC();

// 将媒体关联到对等连接
attachMedia();

// 生成 SDP 提议并发送至对等端
call();


////////////////////
// 以下为函数和处理程序的定义
////////////////////

// 获取本地媒体
function getMedia() {
    // get local audio (microphone)
    navigator.getUserMedia({"audio": true }, function (stream) {
      microphone = stream;
    });

    // 获取本地视频（应用程序分享）
    ///// 这已超出本说明的范围
    ///// 假定“application”已设置此流
    //

// 获取本地视频（前置摄像头）
    constraint =
      {"video": {"mandatory": {"facingMode": "environment"}}};
    navigator.getUserMedia(constraint, function (stream) {
      front = stream;
    });

    // 获取本地视频（后置摄像头）
    constraint =
      {"video": {"mandatory": {"facingMode": "user"}}};
    navigator.getUserMedia(constraint, function (stream) {
      rear = stream;
    });
}
```

```
// 创建对等连接并设置回调
function createPC() {
    pc = new RTCPeerConnection(configuration);

    // 向对等端发送各个 ICE 候选项
    pc.onicecandidate = function (evt) {
       signalingChannel.send(
         JSON.stringify({ "candidate": evt.candidate }));
    };

    // 处理添加的远端流
    pc.onaddstream =
      function (evt) {handleIncomingStream(evt.stream);};
}

// 将媒体关联到 PC
function attachMedia() {
    // 创建要发送的流
    presentation =
      new MediaStream(
        [microphone.getAudioTracks()[0],     // Audio
         application.getVideoTracks()[0]]); // Presentation
    presenter =
      new MediaStream(
        [microphone.getAudioTracks()[0],     // Audio
         front.getVideoTracks()[0]]);        // Presenter
    demonstration =
      new MediaStream(
        [microphone.getAudioTracks()[0],     // Audio
         rear.getVideoTracks()[0]]);         // Demonstration

    // 将流添加到对等连接
    pc.addStream(presentation);
    pc.addStream(presenter);
    pc.addStream(demonstration);

    // 在 SDP 协商之前，即在媒体流开始传输之前
    // 将媒体流 ID 作为 JSON 字符串发送到对等端
    signalingChannel.send(
      JSON.stringify({ "presentation": presentation.id,
                       "presenter": presenter.id,
                       "demonstration": demonstration.id
      }));
}

// 通过创建并发送 SDP 提议发起呼叫
function call() {
    // 请注意，此时我们尚未开始媒体
    // 提议 / 应答过程，因此没有媒体流在传输
```

```
    // 基于当前的流和 ICE 候选项创建 SDP 提议
    // 处理提议时，将调用 gotDescription()
    pc.createOffer(gotDescription);

    // 此函数基于浏览器刚刚创建的 SDP 提议操作
    function gotDescription(desc) {
        // 首先，向浏览器指示此 SDP 提议是我的本地会话描述
        pc.setLocalDescription(desc);

        // 将此提议作为 JSON 字符串发送至对等端
        signalingChannel.send(JSON.stringify({ "sdp": desc }));
    }
}

// 在远端流出现时执行一些处理
function handleIncomingStream(s) {
  // 保存所有传入流的句柄。对于 av_stream，予以呈现
  if (s.getVideoTracks().length == 1) {
    // 这是 av_stream
    av_stream = s;
    show_av(av_stream);
  } else if (s.getAudioTracks().length == 2) {
    // 这是立体声流
    stereo = s;
  } else {
    // 这是单声道流
    mono = s;
  }
}

// 通过将流关联到相应元素显示 / 播放它们
function show_av(s) {
  // display 是视频元素，left 和 right 是音频元素
  display.src = URL.createObjectURL(
                  new MediaStream(s.getVideoTracks()[0]));
  left.src = URL.createObjectURL(
                  new MediaStream(s.getAudioTracks()[0]));
  right.src = URL.createObjectURL(
                new MediaStream(s.getAudioTracks()[1]));
}

// 处理从对等端传入的消息。它们将是 SDP 或 ICE 候选项
signalingChannel.onmessage = function (msg) {
    // 先分析 JSON 事件数据，将其还原为一个对象
    var signal = JSON.parse(msg.data);

    if (signal.sdp) {
        // 如果这是来自对等端的 SDP，则向浏览器指示它是远程端的会话描述
        pc.setRemoteDescription(
          new RTCSessionDescription(signal.sdp));
```

```
    } else {
        // 否则，是来自对等端的候选项
        // 向浏览器指示它是媒体可用于访问对等端的候选 IP 地址
        // 浏览器随后将使用 ICE 来尝试访问此地址
        pc.addIceCandidate(
          new RTCIceCandidate(signal.candidate));
    }
};
```

2.3.2 针对笔记本电脑浏览器的伪码

```
//////////////////
//  这是一段伪代码。没错，它看起来很像真正的代码
//  请不要被迷惑
//  不要认为它能够在某处运行
//  本书接下来的几个章节将提供实际代码
//////////////////

var pc;
var configuration =
  {"iceServers":[{"url":"stun:198.51.100.9"},
                 {"url":"turn:198.51.100.2",
                  "credential":"myPassword"}]};
var webcam, left, right;
var av, stereo, mono;
var incoming;
var speaker, win1, win2, win3;

var signalingChannel = createSignalingChannel();
////////////////////////////////////////////////////////////////////
//  步骤 0 用于建立信令通道。关于如何实现这一点，没有限制要求
//  对等端的身份在建立信令通道期间确定
//  因此，RTCPeerConnection 本身并不包含任何指示对等端身份的配置信息
//  下面的初始代码片段假定此信令通道具有 send() 方法和 onmessage 处理程序
//  前者将其参数发送至对等端，并在对等端触发 onmessage 处理程序执行
////////////////////////////////////////////////////////////////////

// 在这一端，我们基本上是等待 SDP 提议传入，
// 然后建立对等连接、获取本地媒体流并关联它

// 实际上并不需要应答什么
// 这只是创建 PC 和处理程序来处理传入的流
// 它还会设置媒体并由信令通道的 onmessage 处理程序调用
function prepareForIncomingCall() {
    // 首先，创建对等连接
    createPC();

    // 然后，获取本地媒体流
    getMedia();
```

```
    // 将媒体关联到对等连接
    attachMedia();
}

////////////////////
// 以下为函数和处理程序的定义
////////////////////

// 创建对等连接并设置回调
function createPC() {
    pc = new RTCPeerConnection(configuration);

    // 向对等端发送各个 ICE 候选项
    pc.onicecandidate = function (evt) {
      signalingChannel.send(
        JSON.stringify({ "candidate": evt.candidate }));
    };

    // 处理添加的远程流
    pc.onaddstream = function (evt) {handleIncomingStream(evt.stream);};
}

// 获取本地媒体
function getMedia() {

    // 获取本地视频（网络摄像头）
    navigator.getUserMedia({"video": true }, function (stream) {
      webcam = stream;
    });

// 获取本地音频（左声道）
    // 请注意，“direction”尚未定义为约束
    constraint =
      {"audio": {"mandatory": {"direction": "left"}}};
    navigator.getUserMedia(constraint, function (stream) {
      left = stream;
    });

    // 获取本地音频（右声道）
    // 请注意，"direction" 尚未定义为约束
    constraint =
      {"audio": {"mandatory": {"direction": "right"}}};
    navigator.getUserMedia(constraint, function (stream) {
      right = stream;
    });
}

// 将媒体附加到 PC
function attachMedia() {
    // 创建要发送的流
    av = new MediaStream(
```

```
            [webcam.getVideoTracks()[0],        // 视频
             left.getAudioTracks()[0],          // 左声道
             right.getAudioTracks()[0]]);       // 右声道
    stereo = new MediaStream(
                [left.getAudioTracks()[0],      // 左声道
                 right.getAudioTracks()[0]]);   // 右声道

    mono = left;       // 将左声道视为单声道流

    // 将流添加到对等连接
    pc.addStream(av);
    pc.addStream(stereo);
    pc.addStream(mono);

    // 请注意，我们不需要向远程对等端通知流 ID
    // 因为各个流都由其包含的内容唯一标识
}

// 通过创建和发送 SDP 应答来接听呼叫
function answer() {
    // 请注意，此时我们尚未开始媒体提议 / 应答过程
    // 因此没有媒体流在传输

    // 基于浏览器已获取的远程会话描述以及当前的流和 ICE 候选项创建 SDP 应答
    // 处理此应答时，将调用 gotDescription()
    pc.createAnswer(gotDescription);

    // 此函数基于浏览器刚刚创建的 SDP 应答操作。
    function gotDescription(desc) {
        // 首先，向浏览器指示此 SDP 应答是我的本地会话描述
        pc.setLocalDescription(desc);

        // 将此应答作为 JSON 字符串发送至对等端
        signalingChannel.send(JSON.stringify({ "sdp": desc }));
    }
}

// 在远程流出现时执行一些处理
function handleIncomingStream(s) {
  // 只从一路音频流，播放所有视频流
  if (s.id === incoming.presentation) {
    // 使用音频并显示演示屏幕
    speaker.src = URL.createObjectURL(
                    new MediaStream(s.getAudioTracks()[0]));
    win1.src = URL.createObjectURL(
                  new MediaStream(s.getVideoTracks()[0]));
  } else if (s.id === incoming.presenter) {
    // 显示演示者
    win2.src = URL.createObjectURL(
                  new MediaStream(s.getVideoTracks()[0]));
  } else {
```

```
    // 这是演示，予以显示
    win3.src = URL.createObjectURL(
                  new MediaStream(s.getVideoTracks()[0]));
  }
}

// 处理从对等端传入的消息
// 它们将是 SDP、ICE 候选项或包含远程会话 ID 的结构
signalingChannel.onmessage = function (msg) {
    // 首选创建对等连接来应答呼叫
    //（如果尚未应答）
    if (!pc) {
      prepareForIncomingCall();
}
    // 现在分析 JSON 事件数据，将其还原为一个对象
    var signal = JSON.parse(msg.data);

    if (signal.sdp) {
        // 如果这是来自对等端的 SDP，则向浏览器指示它是远程端的会话描述，并发送 SDP 应答
        pc.setRemoteDescription(
          new RTCSessionDescription(signal.sdp));
        answer();
    } else if (signal.candidate) {
        // 如果这是来自对等端的候选项，则向浏览器指示它是可用于访问对等端的候选 IP 地址
        // 浏览器随后将使用 ICE 来尝试访问此地址
        pc.addIceCandidate(new RTCIceCandidate(signal.candidate));
    } else {
        // 这是包含远程对等端将发送的流 ID 的对象。
        // 保存此对象，以便在流出现时使用。
        incoming = signal;
    }
};
```

2.4 参考资料

[XHR] http://www.w3.org/TR/XMLHttpRequest

[XEP-0166] http://xmpp.org/extensions/xep-0166.html

[XEP-0167] http://xmpp.org/extensions/xep-0167.html

[RFC3261] http://tools.ietf.org/html/rfc3261

[SIP-CONNECT] http://www.sipforum.org/sipconnect

[IPSEC] http://tools.ietf.org/html/rfc4301

[EXAMPLE] http://dev.w3.org/2011/webrtc/editor/webrtc.html#examples-and-call-flows

第 3 章 Chapter 3

本地媒体

WebRTC 技术不仅涉及定义如何在两个 WebRTC 对等端之间传输媒体，还涉及定义媒体本身。本章将介绍 WebRTC 的媒体模型以及如何获取和控制本地媒体。后续章节将介绍如何在对等端之间传输媒体。

3.1 WebRTC 中的媒体

3.1.1 轨道

MediaStreamTrack 是 WebRTC 中的基本媒体单元。此轨道代表一种设备或录制内容（称为“源”）可返回的单一类型的媒体。单个立体声源或 6 声道环绕声音频信号均可视为一个轨道，尽管二者都由多个音频声道构成。请注意，规范虽然粗略地规定了各声道“彼此具有公认的关系”，但并未定义在声道级别访问或操作媒体的方式。实际上，根据 WebRTC 文档中的定义，轨道的内容“将一起进行编码，以便作为某种有效负载类型（例如 RTP）进行传输”。换言之，在使用对等连接进行传输时，轨道的各个声道将被视为一个单元，即使其在本地处于启用 / 禁用或静音状态，也不例外。

每个轨道都有一个源与之关联。我们稍后再解释这一现象，但现在要明确一点，即通过 WebRTC 不能直接访问或控制源。对源的一切控制都通过轨道实施。目前，MediaStreamTrack 及其源之间的关系正在发生一些变化，但变化的方向如下：轨道不仅可以是来自源的原始媒体，还可能是浏览器提供的经过转换的版本。例如，轨道可以代表由摄像头以较高本机分辨率录制的视频的低采样率版本。

不同的 MediaStreamTrack 对象可代表同一媒体源，详见以下介绍 MediaStream 对象的章节。有两种方式可用于暂停轨道的媒体：静音和禁用。将轨道静音 / 取消静音这一操作由用户和浏览器执行，而静音则表示轨道的底层媒体源暂时无法提供媒体。例如，如果用户通过单击浏览器 Chrome 中的静音按钮或者切换手机侧面的开关来暂停媒体源的使用权限，就会出现这种情况。一般而言，应用程序无法控制何时将轨道静音。但是，它可以检查轨道的 muted 属性值。静音后，音频轨道将不再发声，视频轨道将显示黑屏。通过将轨道对象的 enabled 属性设置为 false，可单独逐一禁用每个轨道。这两个属性均独立于轨道的 readyState 属性；readyState 属性表示轨道的状态—— new、live 或 ended。如果轨道的状态为 new，则表示其尚未连接至媒体；如果轨道的状态为 ended，则表示其源当前没有且永远无法再提供更多数据。例如，如果拔掉正在使用的摄像头的电源，就会出现这种情况。如果轨道的状态为 live，则表示其可以生成媒体。由于这些属性彼此独立，因而可以同时存在，例如，轨道可同时具有 live、unmuted 和 disabled 属性。

3.1.2 流

MediaStream 是 MediaStreamTrack 对象的集合。有两种方式可用于创建这些 MediaStream 对象：一是通过从现有 MediaStream 中复制轨道来请求对本地媒体的访问；二是使用对等连接来接收新的流。目前，请求和访问本地媒体只有一种方式，即通过调用 getUserMedia()，不过将来可能会出现其他方法，例如从本地文件进行流式传输。通过 MediaStream 构造函数可将现有 MediaStream 对象的轨道“复制”到新的 MediaStream 对象中，该构造函数将现有的一个 MediaStream 对象或一系列 MediaStreamTrack 对象用作参数。还可以完全不指定参数，并能够在现有 MediaStream 中添加轨道（addTrack()）和删除轨道（removeTrack()），但是这一功能尚未得到广泛支持。此外，如果不喜欢将 MediaStream 传递给构造函数，还可以使用 clone() 方法来复制流（及其所有轨道）。请注意，对于派生的媒体流（即基于其他 MediaStream 创建的媒体流），MediaStream 构造函数的数组参数元素不必全都来自现有的同一 MediaStream 对象，而是允许进行混搭。各轨道也可以属于不同的类型，同一 MediaStream 对象可同时包含音频和视频。无论 MediaStream 对象是如何创建的，它们都有一个关键特征，即 MediaStream 对象中的所有轨道都将会在呈现时进行同步。但是，流中的各个轨道并未排序，任何添加重复轨道的尝试都将被忽略，而没有任何提示。由于每个轨道都有一个 ID，且此 ID 会在通过对等连接时得到保留，因此对于通过对等连接发送的 MediaStream，可通过在调用 getAudioTracks() 或 getVideoTracks() 后检查 ID，甚至通过 getTrackById() 直接请求轨道来拼合其中的轨道。与轨道的 ended 状态类似，MediaStream 也有一个布尔型 ended 属性，表示 MediaStream 是否已完成。如果 MediaStream 中所有轨道的状态均为 ended，则将 MediaStream 视为已完成。

3.2 捕获本地媒体

WebRTC 定义了一个新的 JavaScript 方法，专门用于请求对本地媒体的访问：

```
// 请求对音频和视频进行访问
getUserMedia({"audio":true, "video":true},
             gotUserMedia, didntGetUserMedia);
function gotUserMedia(s) {
  console.log(
    "Should be one audio track: " + s.getAudioTracks().length);
  console.log(
    "Should be one video track: " + s.getVideoTracks().length);
}
function gotUserMedia(s) {
  var myVideoElement = getElementById("myvideoelement");

  // 通过视频元素播放捕获的 MediaStream
  myVideoElement.srcObject = s;
}
```

在此示例中，我们请求的媒体流包含一个音频轨道和一个视频轨道。请注意，getUserMedia() 本身并不会返回值。在收到用户许可之前，用户代理不能返回对源的访问（详见图 3.6），而收到用户许可则可能需要数秒或更长时间。为确保整个 JavaScript 线程不终止，可通过回调（本示例中为 gotUserMedia）来返回流。只有在成功获取请求的媒体后，才会调用此回调。否则，将会调用 error 回调。下一节将介绍如何通过约束来选择源，但我们先来看一下源的定义及其使用方式。“媒体捕获和流”规范对此提供了相当细致的说明。源只是理论上的一个实体；浏览器可基于可用的设备以任意方式提供源，而在实际中，物理设备与浏览器提供的源之间可能密切相关。许可绑定到源，每个轨道各自与一个特定的源关联，但同一个源可以有多个关联的轨道。

上面的示例显示了 WebRTC 提供的另一项新功能——添加至媒体元素用于直接赋值的 srcObject 属性。此前，“媒体捕获和流”规范定义了（并建议使用）createObjectURL 来基于 MediaStream 创建 URL，此 URL 可赋予由 getElementById() 返回的 JavaScript 对象的 src 属性。遗憾的是，使用 Blob URL 存在一些问题，最终超出了该规范的范围。现在，该规范为现有的各个媒体元素（例如 <audio> 和 <video>）都定义了新的 srcObject 属性。任何 MediaStream 对象均可直接赋予此属性。

3.3 媒体选择和控制

虽然通过 WebRTC API 不能直接控制源，但可以通过约束来选择源并控制其属性。要了解其工作原理，最简便的途径莫过于先了解现有轨道的操作方式。假设你已获得访问本地视频摄像头的权限：

```
var t;  // 将用于承载轨道

// 请求对视频进行访问
getUserMedia({"video":true},
             gotUserMedia, didntGetUserMedia);
function gotUserMedia(s) {
  t = (s.getVideoTracks())[0];

  // 获取当前功能
  console.log("Capabilites are\n" +
              JSON.stringify(t.getCapabilities() + "\n");

  // 设置约束
  var constraints = {
    "mandatory": {"aspectRatio": 1.3333333333},
    "optional":  [{"width":  {"min": 640}},
                  {"height": {"max": 400}}]
  };
  t.applyConstraints(constraints, successCB, failureCB);
}

function successCB() {
  console.log("Settings are\n" +
              JSON.stringify(t.getSettings() + "\n");
}
```

这可能会在控制台中输出以下内容：

```
Capabilities are
{"width": {"min": 320, "max": 1800},
 "height": {"min": 240, "max": 1200},
 "aspectRatio": {"min": 1, "max": 1.5},
 "facingMode": ["user", "environment"]
}
Settings are
{"width": 640, "height": 480, "aspectRatio": 1.3333333333,
 "facingMode": "user"}
```

可约束的属性分为两种类型：枚举属性或范围属性。枚举属性可以是一组离散字符串值中的任意一个值。范围属性则以最小值 / 最大值的形式给定。调用 getCapabilities() 将返回一个对象，其中包含所有可约束的属性，以及可对其赋予的值（如果是范围属性，则为最小值 / 最大值；如果是枚举属性，则为一个值列表）。调用 getSettings() 将返回所有可约束的属性及其当前值（每个属性一个值）。这里有趣的部分在于对 applyConstraints() 的调用。它用于影响可约束属性的设置。在这里，我们故意使用了“影响”一词。约束机制的目标是提供一种实用的折中状态，既照顾到应用程序开发人员对于控制源参数的需要，又兼顾浏览器（用户代理）实施人员对于“正确行事”的要求。约束机制可视为一种复杂的默认设置系统，它允许应用程序开发人员对事关应用程序的各种细节问题做出规定，同时让浏览

器自主决定该范围之外的事务。例如，浏览器也许可以快速更改编解码器参数来应对不同程度的网络拥塞，但只有应用程序知道何种视频分辨率适合自己。某些应用程序（例如远程外科手术应用程序）或许可以承受较慢的刷新频率，但只有在摄像头的分辨率非常高时才有效。还有一些应用程序（例如足球比赛广播应用程序）或许可以在足球处于空中时承受较低的分辨率，但需要能够快速刷新，以清晰展示足球在足球场上的运动方向。利用约束机制，既可让开发人员控制对应用程序至关重要的参数，又能使浏览器在其他方面智能而灵活地做出选择。

下面介绍约束的工作原理：约束结构（在规范中称为“约束”）是一个对象，它具有两个可选的属性：mandatory 和 optional。可对二者或其中任一属性赋值。对于 mandatory 属性，所赋的值是包含可约束属性的单个对象，必须满足其中每个可约束属性的值，才能成功完成对回调的调用。在该示例中，要求将 aspectRatio 设置为 1.3333333333。aspectRatio 是一个范围属性，可为其设置 min 或 max 值。在此示例中，我们使用所允许的快捷语法来设置单个值。这相当于将 max 和 min 设置为同一个值。对于约束结构中的可选属性，其值和解释并不相同。值是由对象构成的一个数组（有序列表），每个对象均可包含任何需要满足其值的可约束属性。浏览器将尝试满足尽可能多的对象，但即使有任何对象无法得到满足，也不会出错。这些对象的顺序非常重要，如果数组中的两个条目各自均可得到满足，但无法同时满足，则优先满足在数组中排位靠前的对象。实际上，上述示例就存在这种情况。假定该示例 applyConstraints() 调用成功，视频轨道将具有指定的 aspectRatio，因为这是强制性的。这样，轨道的宽度将不小于 640 像素。假定该约束得到满足，则其最小高度必须为 640/1.3333333333 = 480，这就超过了 optional 列表中下一个约束对象所请求的值。这样，如果满足第一个可选约束，就不能满足第二个可选约束。

下面，我们来看一下最初的轨道选择。

```
var constraints = {
  "mandatory": {"aspectRatio": 1.3333333333},
  "optional":  [{"width":  {"min": 640}},
                {"height": {"max": 400}}]
};
getUserMedia({"video": constraints}, successCB, failureCB);
```

我们继续使用上一示例中的相同约束，但这次将其赋予 getUserMedia()。这里的主要差异在于，这些约束用于选择源，而非控制源。getUserMedia() 调用将只返回包含特定轨道的 MediaStream，这些轨道均指向当前可满足所有给定强制性约束的源。返回的轨道将按照给定的约束进行约束。另一个不同点在于，我们可以通过第一个示例中使用的特殊语法来选择源：{“video”:true,“audio”:true}。此语法指示我们希望返回的轨道类型，但没有指定任何约束。

对于任何带有约束的轨道，更改源可能会使源难以满足当前绑定到源的各个轨道的所有约束。例如，如果用户手动更改了设备上的开关来选择或禁用某些配置，就会出现这种

情况。无论在何种情况下，成功获取轨道或成功应用约束之后，如果轨道的约束无法继续得到满足，则将该轨道视为约束过度。如果出现这种情况，浏览器会将该轨道静音，并引发 overconstrained 事件。强烈建议在应用程序中为每个轨道的 onoverconstrained 属性分配一个处理程序，用于应对这一问题。

目前为轨道定义的可约束属性相当少（详见 8.3.2 节），但这些属性当前（或在规范编制完成后）会存储在 IANA 注册表（详见 11.4.1 节）中，以便根据需要定义新的属性。

3.4 媒体流示例

对于图 1.9 中的媒体源，图 3.1 到图 3.4 显示了一种有关轨道和流的安排。在此示例中，图 3.1 和图 3.3 显示了从浏览器 M 流向浏览器 L 的媒体，图 3.2 和图 3.4 显示了反向流动的媒体。需要注意的第一个问题是，媒体流并不对称，两个方向上流动的媒体完全不同。

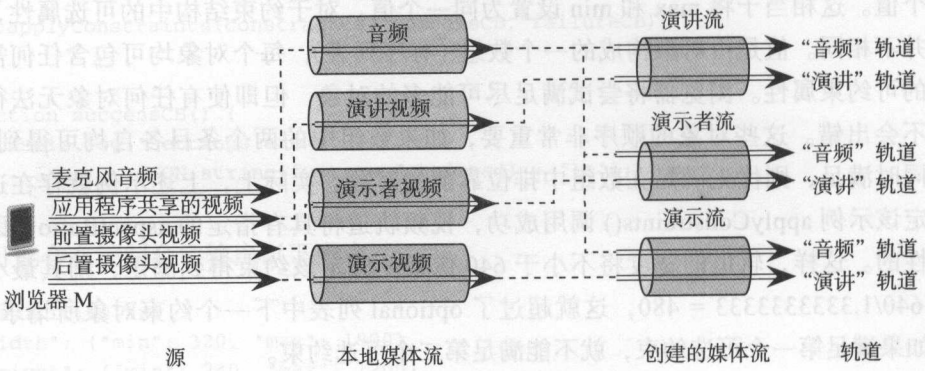


图 3.1 发送源、流和轨道的浏览器 M

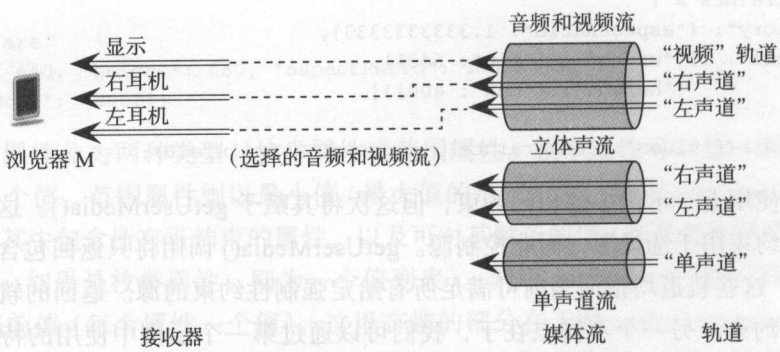


图 3.2 接收轨道、流和接收器的浏览器 M

我们先来看图 3.1，此设备上有四种不同的媒体源可供使用：麦克风音频、应用程序共享视频、前置摄像头视频和后置摄像头视频。通过对 getUserMedia() 分别进行四次调用，

创建了四个单独的本地 MediaStream。随后，JavaScript 应用程序代码通过混搭创建了三个新的 MediaStream 对象。具体而言，第一个 MediaStream 包含从本地音频（麦克风音频源）和演示视频（应用程序共享视频源）MediaStream 中的 MediaStreamTrack 提取的新音频和视频 MediaStreamTrack。

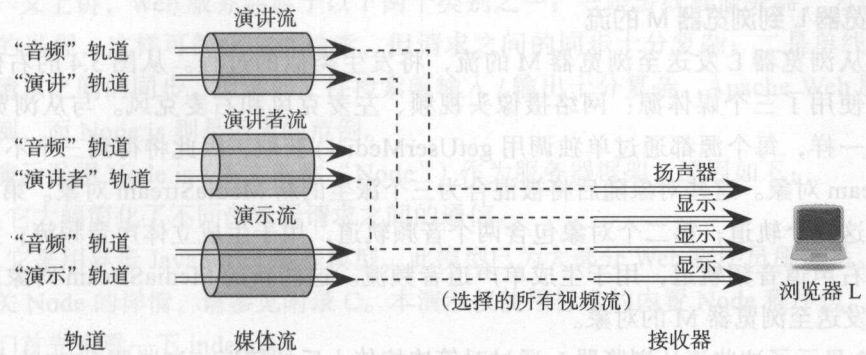


图 3.3 接收轨道、流和接收器的浏览器 L

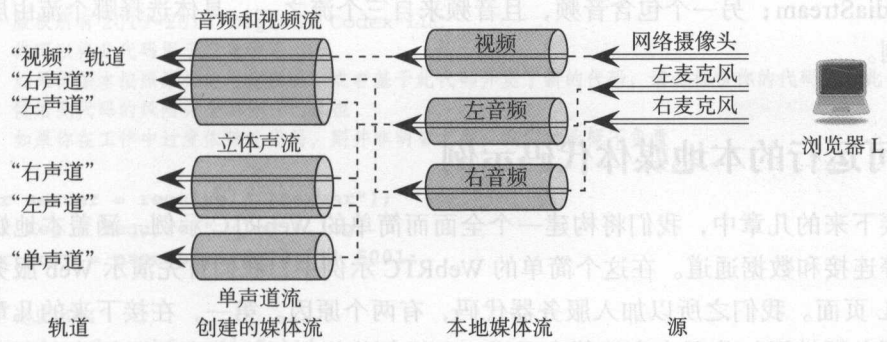


图 3.4 发送源、流和轨道的浏览器 L

与此类似，第二个 MediaStream 包含从本地 MediaStream 的原始内容派生的麦克风音频和前置摄像头视频的新 MediaStreamTrack。最后，第三个 MediaStream 包含从本地 MediaStream 的原始内容派生的麦克风音频和后置摄像头视频的新 MediaStreamTrack。

从浏览器 M 到浏览器 L 的流

现在，应用程序代码有四个本地 MediaStream 对象和三个派生的 MediaStream 对象。请注意，麦克风音频轨道被复制了三次。接下来，应用程序决定将三个派生的 MediaStream 对象发送至远程对等端（位于浏览器 L 中）。图 3.3 显示了这些流从浏览器 M 通过对等连接传入后的情形。请注意浏览器 L 中的应用程序代码接下来如何处理这些流。该代码会执行下列操作：从演示 MediaStream 中提取音频轨道并将其发送至扬声器；从同一 MediaStream 中提取视频轨道并将其发送至屏幕上的一个窗口；从演示者 MediaStream 中提取视频轨道

并将其发送至另一个窗口；从最后一个 MediaStream 中提取视频轨道并将其发送至第三个窗口。虽然图中并未显示，但应用程序可分辨出每一个流，因为每个流都有一个唯一的 ID，并且此 ID 在流通过对等连接时仍会得到保留。当然，需要向浏览器 L 指示演示流、演示者流等的 ID。浏览器 M 中的代码可能会通过信令通道发送此信息。

从浏览器 L 到浏览器 M 的流

对于从浏览器 L 发送至浏览器 M 的流，将发生类似的过程。从图 3.4 的右侧可以看到，总共使用了三个媒体源：网络摄像头视频、左麦克风和右麦克风。与从浏览器 M 到浏览器 L 一样，每个源都通过单独调用 getUserMedia() 获取，因此将得到三个不同的本地 MediaStream 对象。这些对象随后将被混合为三个派生的新 MediaStream 对象。第一个对象包含所有这三个轨道；第二个对象包含两个音频轨道，用于生成立体声音频流；第三个对象只包含右声道音频轨道，用于生成单声道音频流。这些新的 MediaStream 对象就是通过对等连接发送至浏览器 M 的对象。

图 3.2 显示了这些流从浏览器 L 通过对等连接传入后的情形。浏览器 M 中的 JavaScript 应用程序代码决定创建两个新的 MediaStream：一个包含视频，且视频来自包含视频轨道的唯一 MediaStream；另一个包含音频，且音频来自三个流之一，具体选择哪个流由屏幕上的按钮控制。

3.5 可运行的本地媒体代码示例

在接下来的几章中，我们将构建一个全面而简单的 WebRTC 示例，涵盖本地媒体、信令、对等连接和数据通道。在这个简单的 WebRTC 示例中，我们首先演示 Web 服务器和单个 HTML 页面。我们之所以加入服务器代码，有两个原因。第一，在接下来的几章中，我们将在服务器端添加代码来实施信令通道，此示例将提供一个基准，以便读者更容易看到其中的变化。第二，对于许多在线 WebRTC 代码示例，只有注册某种其他的在线服务才能使用，我们认为，提供一个涵盖客户端和服务器的完整示例将有助于你自己进行实验。我们通常在本地计算机上测试各种想法，而根本不需要访问互联网。接下来的两节内容将展示 Web 服务器代码和 HTML 应用程序。前者是完全通用的代码，与 WebRTC API 无关。

3.5.1 Web 服务器

WebRTC 应用程序最简单的形式是 Web 应用程序，这意味着可使用 Web 浏览器通过 Web 服务器来访问它们。一般而言，Web 服务器为 WebRTC 应用程序提供两种实用的功能：一是服务于 Web 应用程序本身；二是充当两个或更多浏览器之间的中继（提供“信令通道”，该中继可用于协商要在浏览器之间传输的媒体的目标、类型和格式。4.5 节将展示基于 HTTP 的简单信令通道的服务器代码。

当今的所有 Web 服务器都能够提供这两项功能，但必须注意的是，WebRTC 应用程序有一个重要的属性要求，即 WebRTC 应用程序都是实时应用程序。由于人们在等待连接成功建立，并且他们已习惯于 PSTN 上的快速呼叫连接，因此如果使用 Web 服务器提供信令通道，就必须高效同步来自要通信的各个浏览器的请求。

从广义上讲，Web 服务器属于以下两个类别之一：一是多线程服务器，为每个请求提供单独的进程，这样可简化文件检索，但请求之间的同步十分复杂；二是单线程服务器，可简化请求之间的同步，但诸如文件检索等输入 / 输出十分复杂。Apache Web 服务器是前者的范例，而 Node.js 则是后者的范例。

本演示采用 Node.js（下文简称“Node”）作为服务器框架，原因如下：

1）它大幅简化了不同浏览器请求之间的通信。

2）它采用异步 JavaScript 编程模型，此模型已为大部分 Web 程序员所熟悉。

有关 Node 的详情，请参见附录 C。本演示仅使用标准的内置 Node 程序包。

我们首先来看一下 index.js。

3.5.1.1 index.js

```
// 版权所有 2013-2014 Digital Codex LLC
// 你可以将此代码用于自身学习
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责

var server = require("./server");
var log = require("./log").log;
var port = process.argv[2] || 5001;

// 返回 404
function fourohfour(info) {
  var res = info.res;
  log("Request handler fourohfour was called.");
  res.writeHead(404, {"Content-Type": "text/plain"});
  res.write("404 Page Not Found");
  res.end();
}

var handle = {};
handle["/"] = fourohfour;

server.serveFilePath("static");
server.start(handle, port);
```

本演示可通过调用“node index.js 5001”来运行，前提是你已安装 Node。

本文件中的代码将执行下列操作：

1）加载“server.js”中的服务器代码和“log.js”中的日志代码。

2）指定将如何处理某些自定义的 URI 路径。

3）指定可处理的静态文件所在的目录。

4）启动 Web 服务器。

下面我们来看看“server.js”中的服务器代码。

3.5.1.2 server.js

```
// 版权所有 2013-2014 Digital Codex LLC
// 你可以将此代码用于自身学习
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责

var http = require("http");
var url = require("url");
var fs = require('fs');

var log = require("./log").log;
var serveFileDir = "";


// 设置静态文件（HTML、JS 等）的路径
function setServeFilePath(p) {
  serveFilePath = p;
}
exports.serveFilePath = setServeFilePath;


// 创建一个处理程序，以基于路径名称来路由请求
function start(handle, port) {
  function onRequest(req, res) {
    var urldata = url.parse(req.url,true),
        pathname = urldata.pathname,
        info = {"res": res};

    log("Request for " + pathname + " received");
    route(handle, pathname, info);
  }

  http.createServer(onRequest).listen(port);

  log("Server started on port " + port);
}

exports.start = start;


// 确定请求的路径是静态文件路径，还是拥有自己的处理程序的自定义路径
function route(handle, pathname, info) {
```

```
  log("About to route a request for " + pathname);
  // 检查前导斜杠后的路径是否为可处理的现有文件
  var filepath = createFilePath(pathname);
  log("Attempting to locate " + filepath);
  fs.stat(filepath, function(err, stats) {
    if (!err && stats.isFile()) {  // 处理文件
      serveFile(filepath, info);
    } else {                       // 必须为自定义路径
      handleCustom(handle, pathname, info);
    }
  });
}

// 此函数先从给定路径名称中删除 ..、~和其他从安全角度而言存在问题的语法位，再向其开头添加 serveFilePath

// ** 我们并没有说此代码现已安全无虞 **
function createFilePath(pathname) {
  var components = pathname.substr(1).split('/');
  var filtered = new Array(),
      temp;

  for(var i=0, len = components.length; i < len; i++) {
    temp = components[i];
    if (temp == "..") continue;     // 没有上级目录
    if (temp == "") continue;       // 没有根目录
    temp = temp.replace(/~/g,'');  // 没有用户目录
    filtered.push(temp);
  }
  return (serveFilePath + "/" + filtered.join("/"));
}


// 打开指定文件、读取其中的内容并将这些内容发送至客户端
function serveFile(filepath, info) {
  var res = info.res;

  log("Serving file " + filepath);
  fs.open(filepath, 'r', function(err, fd) {
    if (err) {log(err.message);
              noHandlerErr(filepath, res);
              return;}
    var readBuffer = new Buffer(20480);
    fs.read(fd, readBuffer, 0, 20480, 0,
      function(err, readBytes) {
        if (err) {log(err.message);
                  fs.close(fd);
                  noHandlerErr(filepath, res);
                  return;}
        log('just read ' + readBytes + ' bytes');
        if (readBytes > 0) {
          res.writeHead(200,
```

```
                                {"Content-Type": contentType(filepath)});
            res.write(readBuffer.toString('utf8', 0, readBytes));
          }
          res.end();
        });
    });
}

// 确定所提取的文件的内容类型
function contentType(filepath) {
  var index = filepath.lastIndexOf('.');

  if (index >= 0) {
    switch (filepath.substr(index+1)) {
      case "html": return ("text/html");
      case "js":   return ("application/javascript");
      case "css":  return ("text/css");
      case "txt":  return ("text/plain");
      default:     return ("text/html");
    }
  }
  return ("text/html");
}

// 确认非文件路径的处理程序，然后执行该程序
function handleCustom(handle, pathname, info) {
  if (typeof handle[pathname] == 'function') {
    handle[pathname](info);
  } else {
    noHandlerErr(pathname, info.res);
  }
}

// 如果没有为请求定义处理程序，则返回 404
function noHandlerErr(pathname, res) {
  log("No request handler found for " + pathname);
  res.writeHead(404, {"Content-Type": "text/plain"});
  res.write("404 Page Not Found");
  res.end();
}
```

在加载所需的 Node 程序包并设置一些变量后，此代码定义了两个要导出的命令——serveFilePath 和 start。第一个命令仅用于设置一个内部变量，对此我们将在后面介绍。

主要方法是 start()。它使用 Node 内置的 HTTP 服务器模块（http.createServer(onRequest).listen（port））来启动侦听端口 5001 的服务器。不过，有趣的地方在于 onRequest 处理程序。要注意的第一个问题是，Node 会对每个请求调用 onRequest 处理程序：向其发送请求（URI）

以及 res 对象（用于承载要发送的响应）。

OnRequest() 先使用 Node 内置的 URL 分析模块来分析请求 URI 中的相关元素，再记录收到的请求，然后调用 route()。

route() 的主要任务是确定该 URI 引用的是要处理的静态文件，还是需要自己特殊处理程序的自定义路径。对于前者，它会调用 serveFile()；对于后者，它会调用 handleCustom。这里有一个有趣而复杂的问题，即如果不检查文件路径，它们可能并不安全。route() 会先调用 createFilePath，再检查是否存在具有该路径名称的文件。

createFilePath() 将检查是否有一些常用路径名称可用于访问正式静态文件目录之外的文件，并删除这些路径名称。请注意，此代码并不保证文件访问的安全性。此类代码的编写和解释已超出本书的范围。但是，Node 中的其他一些模块（例如 Express）可提供更广泛的安全功能。建议读者考虑使用它们。

如前所述，serveFile() 会以异步方式打开请求的文件、读取其中的内容并使用这些内容做出响应。它使用 Node 内置的 fs 模块和 contentType() 帮助函数。

contentType() 会检查文件的扩展名，以便为 HTTP 响应中的 Content-Type 标头确定适当的值。

server.js 中的最后一段代码 handleCustom() 仅尝试定位传入 start() 函数的 handle[] 数组中的自定义路径。如果找到该路径，则执行其中存储的代码。否则，它将使用 noHandlerErr() 返回 HTTP 404。

为确保内容完整，我们来看一下不太重要的“log.js”模块，以便完成服务器端代码的介绍；通过此模块，可以轻松启用或禁用控制台日志记录。

3.5.1.3 log.js

```
// 版权所有 2013-2014 Digital Codex LLC
// 你可以将此代码用于自身学习
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责

var log = console.log;

exports.log = log
```

此处的解释非常简单。简而言之，就是将日志记录到控制台。

3.5.2 客户端 WebRTC 应用程序

```
<!--
// 版权所有 2013-2014 Digital Codex LLC
// 你可以将此代码用于自身学习。
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世
```

```
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责。
-->

<html>
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
  <style>
    video {
      width:  320px;
      height:  240px;
      border:  1px solid black;
    }
    div {
      display:  inline-block;
    }
  </style>
</head>
<body>

<!-- 加载 polyfill，先加载本地副本以进行本地测试 -->
<script src="extra/adapter.js" type="text/javascript"></script>
<script
  src="https://webrtc.github.io/adapter/adapter.js"
  type="text/javascript"></script>

<script>
var myVideoStream, myVideo;

//////////////////////////////
// 这是主例程
//////////////////////////////

// 由此开始获取本地媒体
window.onload = function () {

  myVideo = document.getElementById("myVideo");

  getMedia();
};


//////////////////////////////
// 接下来的这一节代码用于获取本地媒体
//////////////////////////////

function getMedia() {
  getUserMedia({"audio":true, "video":true},
               gotUserMedia, didntGetUserMedia);
}
```

```
function gotUserMedia(stream) {
  myVideoStream = stream;

  // 向我显示我的本地视频
  attachMediaStream(myVideo, myVideoStream);
}

function didntGetUserMedia() {
  console.log("couldn't get video");
}
</script>

<div id="setup">
  <p>WebRTC Book Demo (local media only)</p>
</div>

<br/>

<div style="width:30%;vertical-align:top">
  <div>
    <video id="myVideo" autoplay="autoplay" controls
           muted="true"/>
  </div>
</div>

</body>
</html>
```

在探讨 JavaScript 代码之前，我们先跳至 HTML 标记所在的文件末尾。此应用程序的初始版本只显示一个标题和一个视频元素，几乎再简单不过了。图 3.5 显示了此应用程序在 Firefox 上的视觉效果。

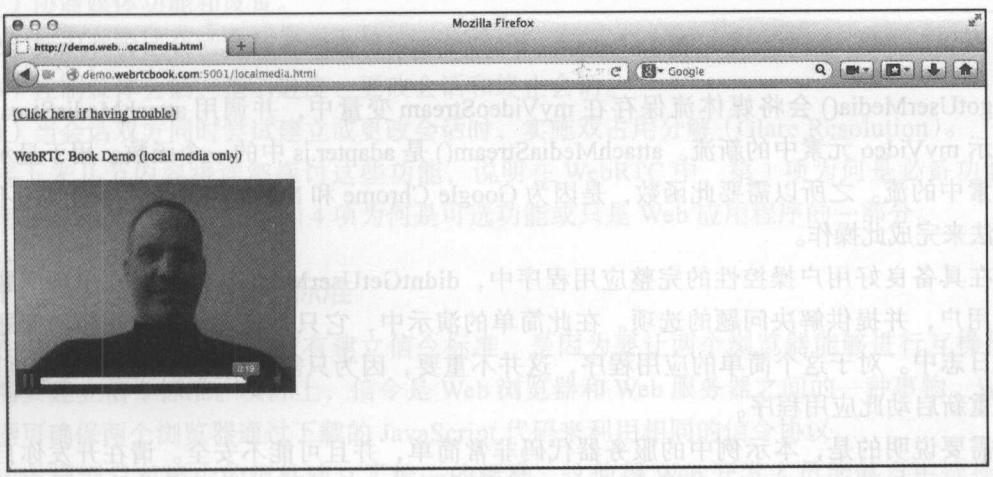


图 3.5 在 Firefox 中仅显示本地媒体

现在，我们回到文件开头。级联样式表（Cascading Style Sheets，CSS）样式仅用于设置视频元素的外观以及 div 元素在屏幕上的布局，我们略去这一内容，直接跳至两个 <script> 行。这两行代码会加载 adapter.js 文件，该文件将标准 API 调用映射至各 Google Chrome 和 Mozilla Firefox 版本当前使用的非标准名称。只有第二行是必需的，因为它引用了该文件的在线位置。该文件由 Mozilla 和 Google 创建。但是，通过将该文件复制到本地，可方便地进行离线测试。如果你已将其复制到本地，则第一行代码将尝试先加载本地副本。

下一 <script> 节包含特定于此简单应用程序的所有 JavaScript 代码。

在此节中，我们先定义所有常用的变量。接下来，先将核心例程定义为将在页面完成加载时执行的函数。随后，此代码将调用 getMedia() 来请求对本地媒体进行访问。

getMedia() 函数非常简单，它会调用 getUserMedia 并请求音频和视频。获取媒体流（需要针对摄像头和麦克风获得用户许可）后，它将使用该流调用 gotUserMedia() 回调。图 3.6 显示了 Firefox 上的许可请求对话框。

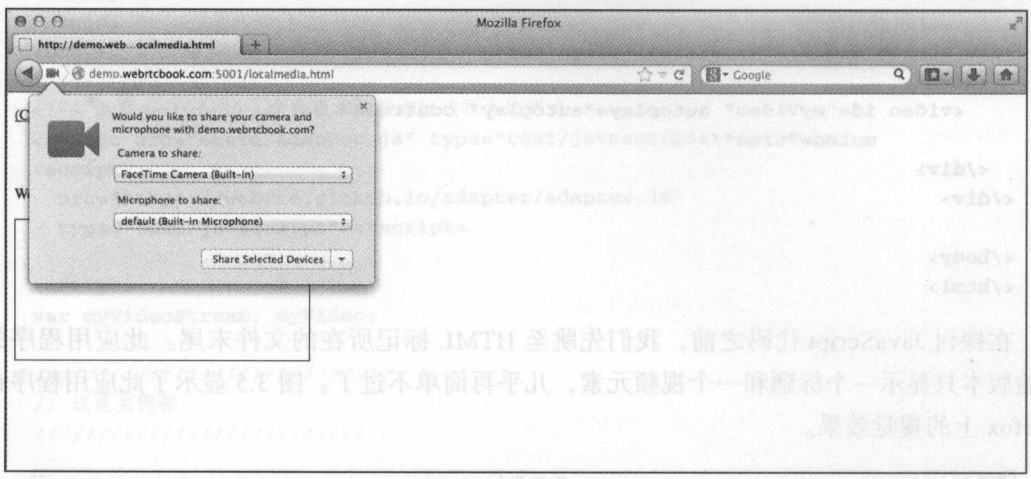
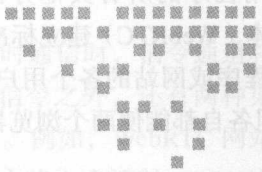


图 3.6 Firefox 中的许可请求对话框

gotUserMedia() 会将媒体流保存在 myVideoStream 变量中，并调用 attachMediaStream() 来显示 myVideo 元素中的新流。attachMediaStream() 是 adapter.js 中的一个函数，用于显示视频元素中的流。之所以需要此函数，是因为 Google Chrome 和 Mozilla Firefox 分别使用不用的语法来完成此操作。

在具备良好用户操控性的完整应用程序中，didntGetUserMedia() 回调会在出现问题时通知用户，并提供解决问题的选项。在此简单的演示中，它只是将无法获取视频的问题记录到日志中。对于这个简单的应用程序，这并不重要，因为只需在浏览器中重新加载页面，即可重新启动此应用程序。

需要说明的是，本示例中的服务器代码非常简单，并且可能不安全。请在开发你自己的服务器代码时多加判断！



第 4 章 Chapter 4

信　令

在 WebRTC 中，信令发挥着举足轻重的作用，但由于没有实现标准化，开发人员可自由选择。这种缺乏统一标准和存在多重选择的局面，让人感觉有些混乱。在实践中，人们提出并使用了多种不同的信令方案，因此了解这些方案之间的差异，将有助于我们在开发 WebRTC 应用程序时做出正确的选择。

4.1 信令的作用

在实时通信中，信令的主要作用体现在四个方面：

1）协商媒体功能和设置。

2）标识和验证会话参与者的身份。

3）控制媒体会话、指示进度、更改会话和终止会话。

4）当会话双方同时尝试建立或更改会话时，实施双占用分解（Glare Resolution）。

接下来几节内容将详细探讨这些功能，说明在 WebRTC 中，第 1 项为何是必备功能，它如何实现标准化，第 2、3 和 4 项为何是可选功能或只是 Web 应用程序的一部分。

4.1.1 为何没有建立信令标准

在 WebRTC 中，之所以没有建立信令标准，是因为要让两个浏览器能够进行互操作，并不需要建立信令标准。实际上，信令是 Web 浏览器和 Web 服务器之间的一种事物。Web 服务器可确保两个浏览器通过下载的 JavaScript 代码来利用相同的信令协议。

Web 模型只对极少的组件建立了统一的标准，这使得 Web 开发人员能够自由选择和

设计网页与应用程序的所有其他方面。实际上，这意味着只需要对传输（HTTP）、标记（HTML）和媒体（WebRTC）建立标准即可。如图 4.1 所示，服务器负责选择信令协议，并确保 Web 应用程序或网站的各个用户支持该协议。Web 服务器 A、B 和 C 不需要使用相同的信令协议，但各自都能使两个浏览器建立媒体会话。

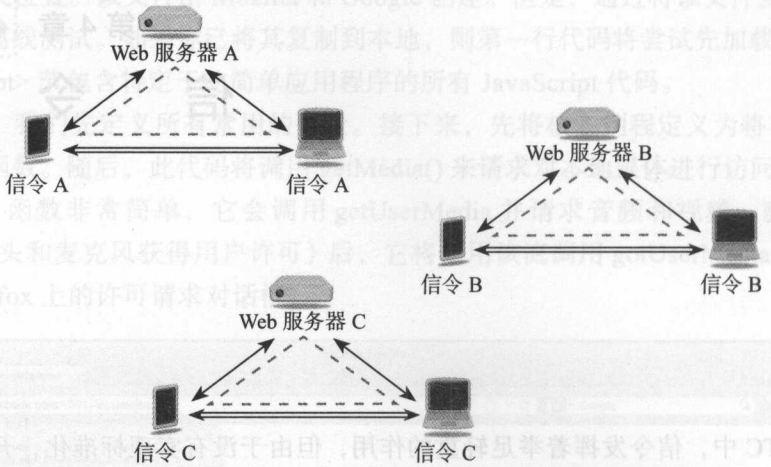


图 4.1 Web 服务器选择信令协议

与此不同的是，在常规 VoIP 或视频系统中，信令或控制服务器无法向终端设备中推送信令代码。因此，实现互操作功能的唯一方式是，让两个终端使用同一标准化信令协议，例如 SIP 或 Jingle（详见 4.3.6 节和 4.3.7 节）。

对于联合或梯形体系结构（例如图 1.5 中显示的体系结构），两个 Web 域需要就信令协议达成一致才能进行互操作。但是，此信令协议不必是每个浏览器中使用的信令协议。换言之，不能只因为两个 Web 域使用 SIP 进行通信，就认为两个浏览器也必定使用 SIP。

4.1.2 媒体协商

WebRTC 规范包含针对“信令通道”的要求。信令最重要的功能在于，在参与对等连接的两个浏览器之间交换会话描述协议（Session Description Protocol，SDP）对象中包含的信息。SDP 包含供浏览器中的 RTP 媒体栈配置媒体会话所需的全部信息，包括媒体类型（音频、视频、数据）、所用的编解码器（Opus、G.711 等）、用于编解码器的各个参数或设置，以及有关带宽的信息。此外，信令通道还用于交换候选地址，以便进行 ICE 打洞。候选地址表示浏览器可从中接收潜在媒体数据包的 IP 地址和 UDP 端口。在信令通道中收发的候选地址也可以不包括在 SDP 中。另外，还必须在信令通道中交换用于 SRTP 的密钥材料。

只有通过信令通道交换候选地址后，才能开始 ICE 打洞（详见 9.2 节），因此如果没有此信令功能，就无法建立对等连接。

4.1.3 标识和身份验证

在使用标准信令协议（例如 SIP 或 Jingle）发起实时通信时，信令通道将提供参与者的标识，并可以选择进行身份验证。在 WebRTC 中，除信令之外，还有两种渠道可用于确定身份。第一种渠道是参见 Web 应用程序提供的上下文。例如，WebRTC 网站的用户可能使用特定的屏幕名称进行登录。当此用户希望与另一用户建立会话时，Web 应用程序就会将其屏幕名称作为标识提供给对方。除了相信该网站提供了准确的标识之外，另一端的用户别无选择。这与公共交换电话网络（Public Switched Telephone Network，PSTN）中的呼叫方标识非常相似。PSTN 用户只能相信其服务提供商向其电话呈现的呼叫方 ID 就是呼叫方本人，而没有其他方式进行独立确认。第二种渠道是查看 URL 中可能传递的标识，此 URL 可能包含随机令牌。在通过这种方式建立的 WebRTC 会话中，双方应该都知道该标识令牌。

WebRTC 定义了另一种标识方法，即通过媒体通道，此方法不依赖于网站信息的可信性。媒体路径标识的概念最早由 ZRTP [RFC6189] 媒体路径密钥协议提出，根据这一概念，呼叫方的标识与身份验证由媒体路径提供，完全不依赖于信令通道。WebRTC 采用 DTLS-SRTP [RFC5763] 提供媒体路径标识。这通过 DTLS 握手期间使用的公钥的指纹实现。此指纹可使用标识提供程序（详见 13.4 节）进行身份验证。信令通道用于传输指纹和标识断言，但不以任何其他形式参与标识断言的生成或验证。

4.1.4 控制媒体会话

传统多媒体信令协议（例如 SIP、Jingle 或某种专有协议）可提供会话呼叫控制。专有协议可能极为简单，例如 4.5 节中演示应用程序就使用了这样的协议。但是，在 WebRTC 中，虽然需要信令才能发起或更改媒体会话，但不需要信令来指示状态或终止会话。实际上，浏览器中的 ICE 状态机可提供这些信息。例如，在检查候选地址时，ICE 状态机可提供有关会话的进度信息。建立会话后，如果 ICE 持续同意检查失败，则表明会话已终止。

4.1.5 双占用分解

SIP 等信令协议内置有双占用分解功能。当通信会话的双方同时尝试建立或更改会话时，就会出现双占用问题。这是一种争用问题，可能导致会话出现无法确定的状态。一些 SDP 使用方案消除了多种双占用情况，如果在 WebRTC 中采用这些方案，将大大减少双占用问题引发的处理需求。例如，如果无需进行新的提议 / 应答交换，即可向会话中添加新的媒体源，那么就可以消除这一常见的双占用问题的触发因素。

4.2 信令传输

WebRTC 要求在两个浏览器之间建立双向信令通道。我们先来讨论信令消息的传输。

通常有三种方式用于传输 WebRTC 信令：HTTP、WebSocket 和数据通道。

4.2.1 HTTP 传输

HTTP 还可用于传输 WebRTC 信令。浏览器可发起新的 HTTP 请求，以便向服务器发送信令信息并从中接收信令信息。信令信息可使用 GET 或 POST 方法或以应答形式传输。如果信令服务器支持跨域资源共享（Cross-Origin Resource Sharing，CORS）（详见 13.2.3 节），则其 IP 地址可以不同于 Web 服务器。

向服务器发送信息时，将使用 JavaScript 中的 XML HTTP 请求（XML HTTP Request，XHR）API 调用，操作非常简单。反向而言，从服务器异步接收信息则较为复杂；对此，近年来开发出了一系列技术，统称为异步 AJAX（Asynchronous JavaScript And XML）[AJAX]。其中一些方案非常简单，它们单纯通过轮询或让 GET 请求持续开放来准备接收来自服务器的数据；例如，4.5 节中的演示应用程序代码使用的就是这种信令方案。一般情况下，我们通过 JavaScript 库（例如 jQuery [JQUERY]）来间接使用 XHR。4.5 节中的演示应用程序对信令通道采用 HTTP 传输模式。

图 4.2 显示了 HTTP 传输模式的用法。请注意，使用 HTTP 传输信令的做法通常称为表述性状态传输（Representational State Transfer，REST）或 RESTful 信令。

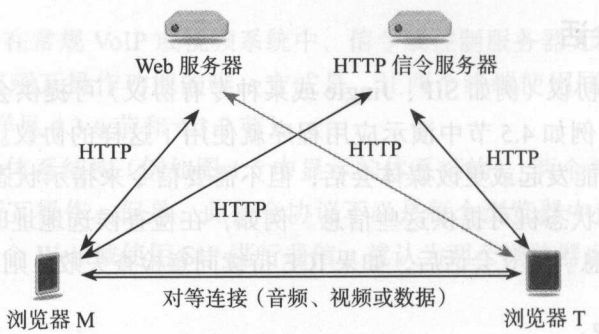


图 4.2 使用 HTTP 传输信令

HTTPS 比 HTTP 更为安全，它允许浏览器对信令服务器进行身份验证，详见 13.1.2 节。

4.2.2 WebSocket 传输

WebSocket 传输（详见 10.2.2 节）允许浏览器开通一个与服务器的双向连接。此连接最初采用 HTTP 请求形式，但随后会升级为 WebSocket。只要服务器支持 CORS，WebSocket 服务器的 IP 地址就可以不同于 Web 服务器，如图 4.3 所示。虽然此图中显示的是 SIP 代理服务器，但它支持 WebSocket。任何信令都可以通过 WebSocket 进行传输，包括 4.5 节的演示中使用的简单信令。

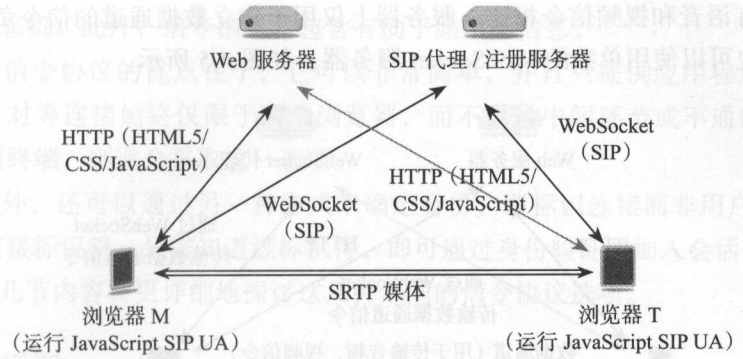


图 4.3 使用 WebSocket 传输信令

为了能够访问 WebSocket 服务器，它必须拥有公共 IP 地址，并且运行 HTTP 服务器。请注意，这意味着无法直接与另一个浏览器开通 WebSocket，因为浏览器实施的是 HTTP 用户代理功能而非 HTTP 服务器功能，因此仍然需要 WebSocket 服务器在使用 WebSocket 的两个 Web 客户端之间提供中继。即使计算机运行的是 HTTP 服务器，NAT 遍历也会阻止在 Internet 上的两台计算机之间建立端到端 WebSocket 连接。浏览器通过 WebSocket API [WS-API] 来使用 WebSocket。

所有主流浏览器供应商都支持 WebSocket。但是，某些 Web 代理和防火墙并不完全支持 WebSocket 并可能导致问题，尤其是在身份验证方面。

4.2.3 数据通道传输

当在两个浏览器之间建立数据通道后，它就会提供直接的低延迟连接，这使其非常适合用于传输信令。由于最初建立数据通道时需要单独的信令机制，因此数据通道无法单独用于传输所有 WebRTC 信令。但是，一旦建立数据通道，即可使用它来处理所有信令，包括用于通过对等连接传输的音频和视频媒体的所有信令。图 4.4 显示了这一情况，其中数据通道信令通过 HTTP 传输到 Web 服务器，而所有其他信令则通过数据通道传输。

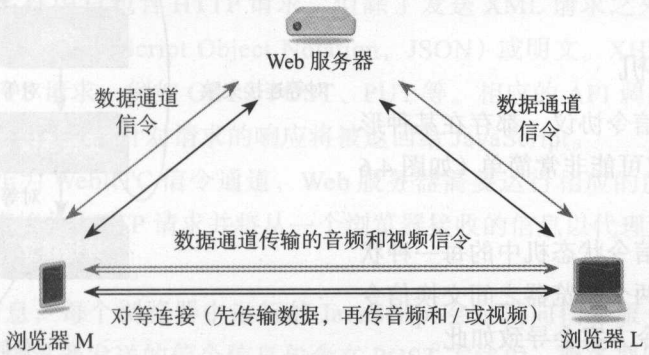


图 4.4 使用数据通道传输信令

与处理所有语音和视频信令相比，服务器上仅用于建立数据通道的信令负载要少得多。数据通道信令也可以使用单独的 WebSocket 服务器，如图 4.5 所示。

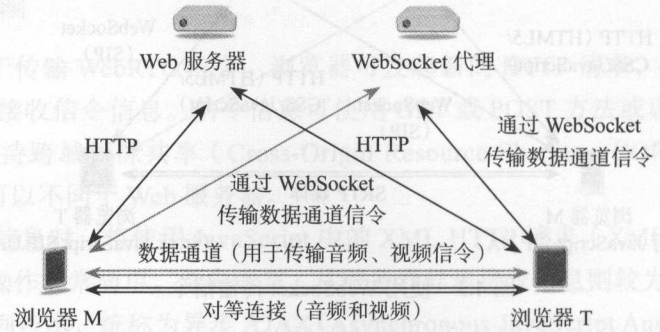


图 4.5　通过单独服务器传输数据通道信令

利用数据通道传输信令的一个有趣的好处在于，它有助于缩短用户感知的建立连接所需的时间。在浏览器之间交换媒体之前，必须先完成 ICE 和 DTLS-SRTP 密钥管理步骤。如果等到被叫方用户接受会话（即“应答呼叫”）后才开始执行这些步骤，则会出现明显的延迟，最长可能达到几秒。利用数据通道传输信令时，将执行 ICE 和 DTLS 密钥管理来建立数据通道，这可以发生在提示或要求用户接受会话之前。当用户接受会话后，将能够以相当快的速度添加语音和视频，因为信令消息将直接传输至另一端的浏览器，媒体流可重复使用对等连接，而无需执行 ICE 或 DTLS-SRTP 处理。

4.3　信令协议

WebRTC 信令协议的选择至关重要，而且不必局限于所选的信令传输方式（详见上一节）。开发人员可选择创建自己的专有信令协议，采用 SIP 或 Jingle 等标准信令协议，或者使用通过抽象化处理剥离了信令协议细节的库。

4.3.1　信令状态机

无论采用何种信令协议，都存在某种形式的“状态机”，它可能非常简单（如图 4.6 所示）。

请注意，对于信令状态机中的每一种状态，并非都需要在两个浏览器之间交换信令消息，尽管某些信令方案会导致如此。

采用标准信令协议的优点在于，已存在

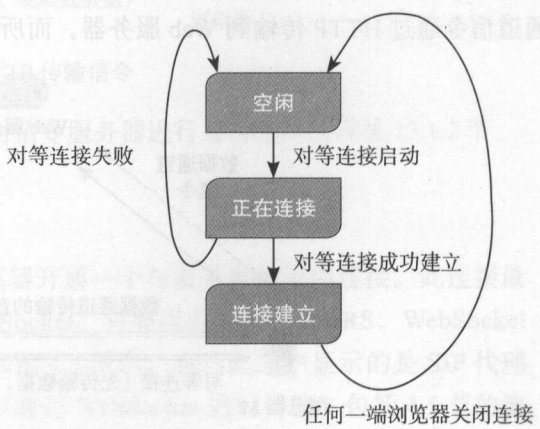


图 4.6　简单的信令状态机

完全定义的状态机。此外，信令消息将包含有助于路由的信息。

采用专有信令协议的优点在于，它可以非常简单，并且只提供应用程序所需的功能。如果 WebRTC 对等连接始终仅限于两个浏览器，而不通过中间环节或不通向 SIP 或 Jingle VoIP 或是视频终端，则适合采取这一选项。

除信令之外，还可以通过另一种方式来确定身份，即标识连接而非用户。例如，可在 URL 中加入连接标识符，只需知道该标识符，即可通过身份验证并加入会话。

接下来的几节内容将更详细地探讨这几种不同的信令协议选项。

4.3.2 信令标识

为实现 4.1 节指出的第 2 项作用，需要在服务器中设置某种路由逻辑。如果浏览器和服务器之间的给定连接可通过令牌标识，则发送至服务器的信令消息可包含另一个“服务器至浏览器”连接的令牌。这样，Web 服务器代码将在两个连接之间充当代理或用于转发信息。有两个例子可用于说明此方案：一是使用 WebSocket 代理，二是使用 Google 应用程序引擎通道 API。如果采用标准信令协议，消息将包含对等端浏览器（用户）的标识符作为目标。对于这种特定的信令协议，可通过标准服务器来提供此功能，而且只需极少的配置。例如，如果采用 SIP 信令，可通过 SIP 代理服务器执行此功能。如果采用 Jingle，可通过 XMPP 服务器提供此功能。

4.3.3 HTTP 轮询

HTTP 轮询是一种简单的专有信令方案。例如，本章后面介绍的基于 XHR 的信令通道就属于 HTTP 轮询。

通过在 JavaScript 或 jQuery 中调用 XHR，可使 JavaScript 应用程序针对 Web 服务器生成新的 HTTP 请求并处理 HTTP 响应。XHR 是一种 W3C 标准 API，当前定义为工作草案 [XHR]，XHR 正逐步成为一项标准。XHR JavaScript API 的各个功能组件均受浏览器支持。虽然 XHR 的名称中只包含 HTTP 请求，但除了发送 XML 请求之外，它还可以发送 JavaScript 对象表示法（JavaScript Object Notation，JSON）或明文。XHR 可使浏览器生成新的 HTTP 或 HTTPS 请求，例如 GET、POST、PUT 等。相应的 API 调用将指定要使用的方法以及 IP 地址和端口号。针对请求的响应将被返回给 JavaScript。

要使用 XHR 作为 WebRTC 信令通道，Web 服务器需要运行相应的应用程序，用于通过另一个 XHR 通道接收 HTTP 请求并将从一个浏览器接收的信息以代理形式发送或转发给另一个浏览器，如图 4.7 所示。

为交换信令信息，每个浏览器中运行的 JavaScript 会定期向信令服务器发送 HTTP 消息，以进行轮询。浏览器发送的信令信息包含在 POST 方法中。服务器收到的信令信息包含在发送给 POST 的 200 OK 响应中。请注意，在此方案中，每个消息都是一个新请求。

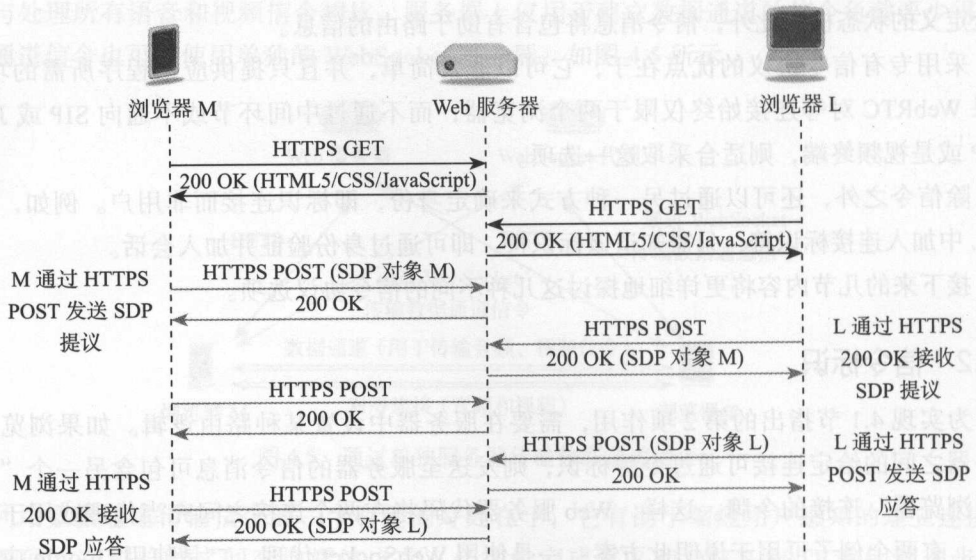


图 4.7 使用 Web 服务器 HTTP 轮询传输信令

4.3.4 WebSocket 代理

对于用来传输 WebRTC 信令的 WebSocket 代理，其使用的服务器将具有公共 IP 地址，并可由建立对等连接的两个浏览器访问。每个浏览器将分别与该服务器开通一个独立的 WebSocket 连接，服务器则将两个连接桥接在一起，以代理身份将信息从一个浏览器发送至另一个浏览器，如图 4.8 所示。由于 JavaScript 并不支持 DNS 查找，因此 Web 服务器将需要以 IP 地址和端口号的形式来提供 WebSocket 服务器。

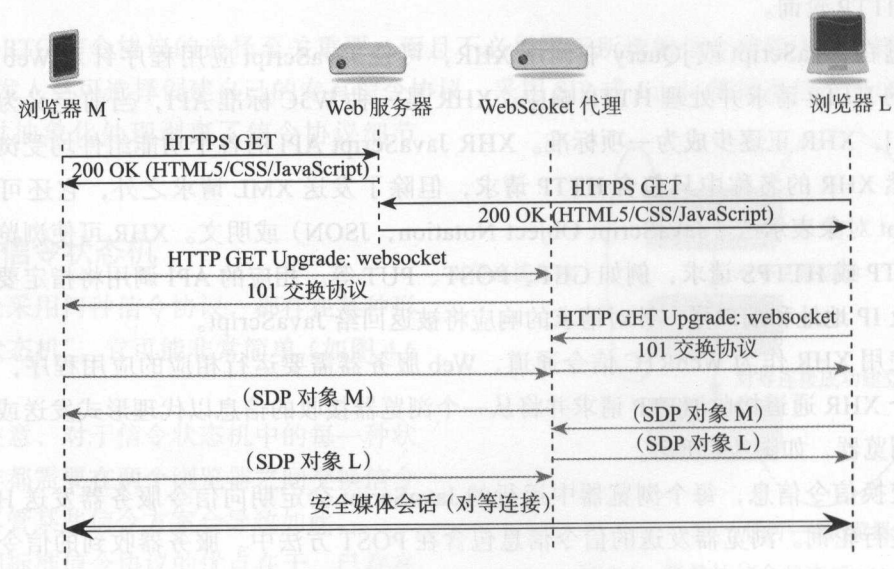


图 4.8 信令示例：WebSocket 代理

[GINGER-TECH] 中提供了这样一个示例。此示例创建了一个简单的 WebSocket 服务器，用于侦听 WebSocket 连接的特定端口。当收到来自 WebSocket 的信息时，该服务器会将其广播到所有开通的 WebSocket 连接。对于实际 WebSocket 代理，服务器不会广播至所有开通的连接，而只限于对等端浏览器所在的特定连接。为此，需要某种形式的会话 ID。另一种简单的方案是为每个对等连接分配 WebSocket 代理上的一个端口。端口将随机分配并在各浏览器之间共享。

下面是其中的一些信令伪代码片段：

```
// 定义信令通道
function createSignalingChannel(msgHandler) {
  var socket = new WebSocket('wss://192.168.0.1:49152/');
  socket.addEventListener("message", msgHandler, false);
  return {"send": socket.send};

}

// 打开 WebSocket
signalingChannel = createSignalingChannel(onMessage);

// 通过 WebSocket 发送消息
signalingChannel.send(message);

// 处理通过 WebSocket 传入的消息
function onMessage(msg) {
// 处理消息
}
```

通过采用安全的 WebSocket 传输方式，可在浏览器和 WebSocket 服务器之间提供加密。

4.3.5 Google 应用程序引擎通道 API

另一种常见而专有的方案是使用 Google 应用程序引擎通道 API 作为信令通道，详见 [APPRTC]。利用 Google 应用程序引擎，JavaScript 客户端可与 Google 云中的服务器建立通道。此通道 API 使用 XHR 将来自浏览器的信令消息发送至 Google 服务器。它使用通称为 Comet 的技术将消息转发给另一个浏览器。Comet 采用 HTTP 长轮询技术，又称为 AJAX、AJAX 推送或 HTTP 服务器推送。这通常涉及让客户端向服务器发送一个请求，并使该请求保持打开状态，直至服务器向客户端发送信息。（Comet 名称具有双关含义，因为它还是一种清洁剂的品牌，这一点与 Ajax 类似。）服务器通过应用程序引擎通道接收消息。

通过使用唯一的客户端 ID，可将信息发送至服务器，再转发给另一端的客户端。有关如何对 Java 使用该通道 API 的概述，请参见此处 [APP、ENGINE-CHANNEL]。另一端的浏览器由客户端 ID 标识。

调用 goog.appengine.Channel（channelToken）可打开该通道。应为以下事件设置事件

处理程序：onopen、onmessage、onerror、onclose。请参见如下 XHR 伪代码片段：

```
function createSignalingChannel(channelToken, msgHandler) {
  var channel = new goog.appengine.Channel(channelToken);
  var handler = {
    'onopen': onChannelOpen,
    'onmessage': msgHandler,
    'onerror': onChannelError,
    'onclose': onChannelClose
  };
  var socket = channel.open(handler);

  function onChannelOpen() {
          // 通道打开
  }
  function onChannelError() {
          // 通道出错
  }
  function onChannelClosed() {
          // 通道关闭
  }

  function XHRSend(msg) {
    var req = new XMLHttpRequest();
    req.open('POST', /* 服务路径 */, true);
    req.send(msg);
  }

  return {'send': XHRSend};
}

// 打开通道
signalingChannel =
  createSignalingChannel(/* 此处为某种令牌 */, onMessage);

// 通过 XHR 发送消息
signalingChannel.send(message);

// 处理来自应用程序引擎通道的传入消息
function onMessage(msg) {
  // 处理消息
}
```

4.3.6 WebSocket SIP

WebSocket SIP 是另一种方案。会话初始协议 SIP[RFC3261] 是一种信令协议，通常用于 IP 语音（Voice over IP，VoIP）和视频会议系统。SIP 是服务提供商 IP 多媒体子系统（IP Multimedia Subsystems，IMS）中的关键协议，也是用于 PSTN 更换的 SIP 中继。此外，

SIP 还在企业通信系统中用于统一通信（Unified Communications，UC）以及即时消息传递（Instant Messaging，IM）和联机状态。SIP 可使用 UDP、TCP、SCTP 或 TLS 作为传输机制。新的 RFC 定义了 WebSocket SIP 传输方案 [RFC7118]。

在此信令方案中，浏览器先加载 JavaScript SIP 用户代理，然后与支持 WebSocket 传输的 SIP 代理 / 注册服务器建立连接（REGISTER）。发起 WebRTC 会话的浏览器将向代理服务器发送一条包含 SDP 提议的 INVITE 消息。目标浏览器将由 SIP URI（例如 sip:user@webserver.org）标识。另一端的浏览器将收到 INVITE 并生成相应的 SIP 响应，以通过 200 OK 响应返回 SDP 应答，从而建立媒体会话。

表 4.1 列出了当前支持 WebSocket SIP 的一些开源 SIP 栈。此外，许多商业 SIP 栈也支持 WebSocket SIP。

表 4.1 开源 WebSocket SIP 实现方案

协议栈	角色	URL
JsSIP	JavaScript SIP 用户代理	http://jssip.net
sipML5	JavaScript SIP 用户代理	http://sipml5.org
WebRTComm	JavaScript SIP 用户代理	https://code.google.comp/webrtcomm
Asterisk	支持 WebSocket 的 SIP 服务器	http://asterisk.org
OverSIP	支持 WebSocket 的 SIP 服务器	http://oversip.net
Kamailio	支持 WebSocket 的 SIP 服务器	http://kamailio.org
OfficeSIP	支持 WebSocket 的 SIP 服务器	http://officesip.com

图 4.9 显示了通过 WebSocket SIP 传输 WebRTC 信令时的消息流。

下面显示了一条示例 WebSocket SIP 消息：

```
INVITE sip:bob@atlanta.com SIP/2.0
  Via: SIP/2.0/WS df7jal23ls0d.invalid;branch=z9hG4bK56sdasks
  From: sip:alice@atlanta.com;tag=asdyka899
  To: sip:bob@atlanta.com
  Call-ID: asidkj3ss
  CSeq: 1 INVITE
  Max-Forwards: 70
  Supported: path, outbound, gruu
  Route: <sip:proxy.atlanta.com:443;transport=ws;lr>
  Contact: <sip:alice@atlanta.com
   ;gr=urn:uuid:f81-7dec-14a06cf1;ob>
  Content-Type: application/sdp

  (SDP Offer not shown)
```

请注意 Via 标头字段中的主机名称（该名称为 .invalid，这导致其不可路由）、Via 中的 WS 令牌，以及 Route 标头字段中的 transport=ws 参数。WebSocket SIP 使用 SIP Path 标头字段 [RFC3327]、SIP 出站机制 [RFC5626] 和 GRUU [RFC5627]。

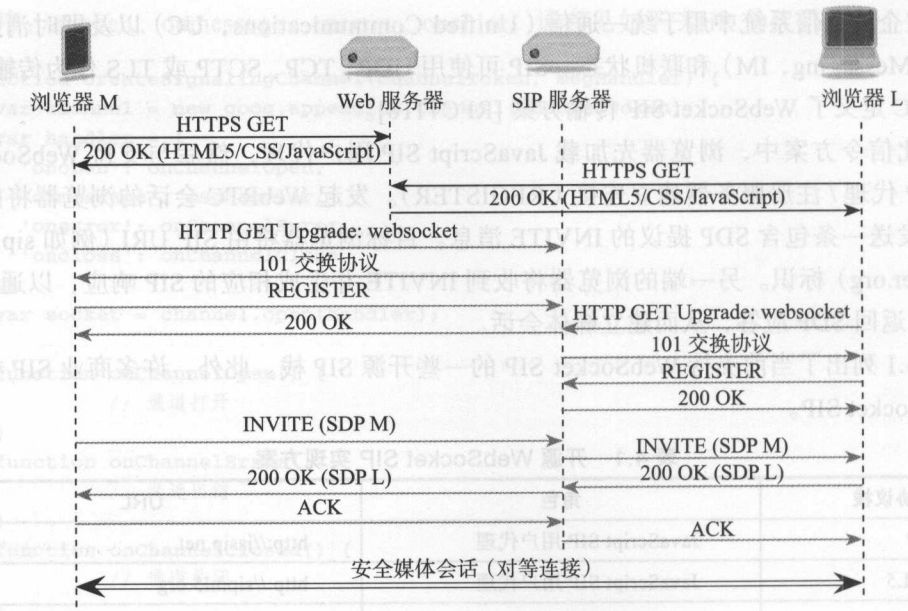


图 4.9 WebSocket SIP 信令

请参见下面的示例 jsSIP 伪代码：

```
// 初始化 SIP UA
var configuration = {
  'outbound_proxy_set': 'ws://sip-ws.example.com',
  'uri': 'sip:alice@example.com',
  'password': '123456'
};

var sipUa = new JsSIP.UA(configuration);

sipUA.call('sip:bob@example.com',useAudio, useVideo, eventHandlers, views);
```

4.3.7 WebSocket Jingle

Jingle 是对可扩展消息传递和联机状态协议（extensible Messaging and Presence Protocol，XMPP）的扩展，又称为 Jabber [RFC6120]，它向 XMPP 中添加了媒体信令功能。利用 Jingle，可将 SDP 会话描述映射至 XML 格式，再通过 TCP 或 TLS 传输至 XMPP 服务器。独立 XMPP Jabber 客户端通常用于即时消息传递和联机状态，GoogleTalk 和其他企业 IM 服务都使用这种客户端。多年来，XMPP 一直通过多种技术嵌入在网页之中，其中包括异步 HTTP 双向流（Bidirectional-streams Over Synchronous HTTP，BOSH）[XEP- 0124]，此技术可用作 XMPP 传输机制 [XEP-0206]。有一种新的 Internet 草案定义了通过 WebSocket 传输 XMPP 的方案 [draft-ietf-xmpp-Websocket]，这种方案有望提供更好的性能和互操作性。

为使用 Jingle 实现 WebRTC 信令通道，需要从网页中下载并运行以 JavaScript 编写的 Jingle XMPP 客户端。该客户端将通过 WebSocket 建立通向 XMPP 服务器的 XMPP 连接。随后，该客户端会将浏览器生成的 SDP 提议和应答映射到 Jingle 调用设置消息中，并将它们转发给另一端的浏览器。另一端的浏览器将由 Jabber ID（JID）标识。图 4.10 显示了 WebRTC 中的 Jingle 信令消息流。

表 4.2 列出了 JavaScript XMPP 栈。

表 4.2 [XMPP-LIBRARIES] 中的开源 XMPP 栈

协议栈	角色	URL
Strophe	使用 BOSH 的 JavaScript XMPP 栈	http://strophe.im/
node-xmpp	使用 BOSH 和 WebSocket 的 JavaScript XMPP 栈	https://github.com/astro/node-xmpp
dojox-xmpp	使用 BOSH 的 JavaScript XMPP 栈	http://dojotoolkit.org/api/1.3/dojox
frabjous	JavaScript XMPP 栈	https://github.com/theozaurus/frabjous
jQuery-XMPP-plugin	JavaScript XMPP 栈	https://github.com/maxpowel/jQuery-XMPP-plugin

下面显示了一条示例 Jingle 消息：

```
<iq id="3Tzpd-1650" to=bob@example.com/jitsi-3u1kluu
    from=" alice@example.org/jitsi-2rbmomp" type="set">
  <jingle xmlns='urn:xmpp:jingle:1'
          action='session-initiate'
          initiator=' alice@example.org/jitsi-2rbmomp'
          sid='3pd4ihhk9t72q'>
    <content creator='initiator' name=' audio'>
      <description xmlns='urn:xmpp:jingle:apps:rtp:1'
                   media='audio'>
        <payload-type id='96' name=' opus' channels= '1'
                      clockrate='16000'/>
        <payload-type id='0' name='PCMU' channels= '1'
                      clockrate= '8000' />
      </description>
      <transport xmlns='urn:xmpp:jingle:transports:ice-udp:1'
                 pwd='asd88fgpdd777uzjYhagZg' ufrag='8hhy'>
        <candidate component='1' foundation='1' generation='0'
                   id='el0747fg11' ip='10.0.1.1' network='1'
                   port='8998' priority='2130706431'
                   protocol='udp' type='host'/>
        <candidate component='1' foundation='2' generation='0'
                   id='y3s2b30v3r' ip='192.0.2.3' network='1'
                   port='45664' priority='1694498815'
                   protocol='udp' rel-addr='10.0.1.1'
                   rel-port='8998' type='srflx'/>
      </transport>
    </content>
  </jingle>
</iq>
```

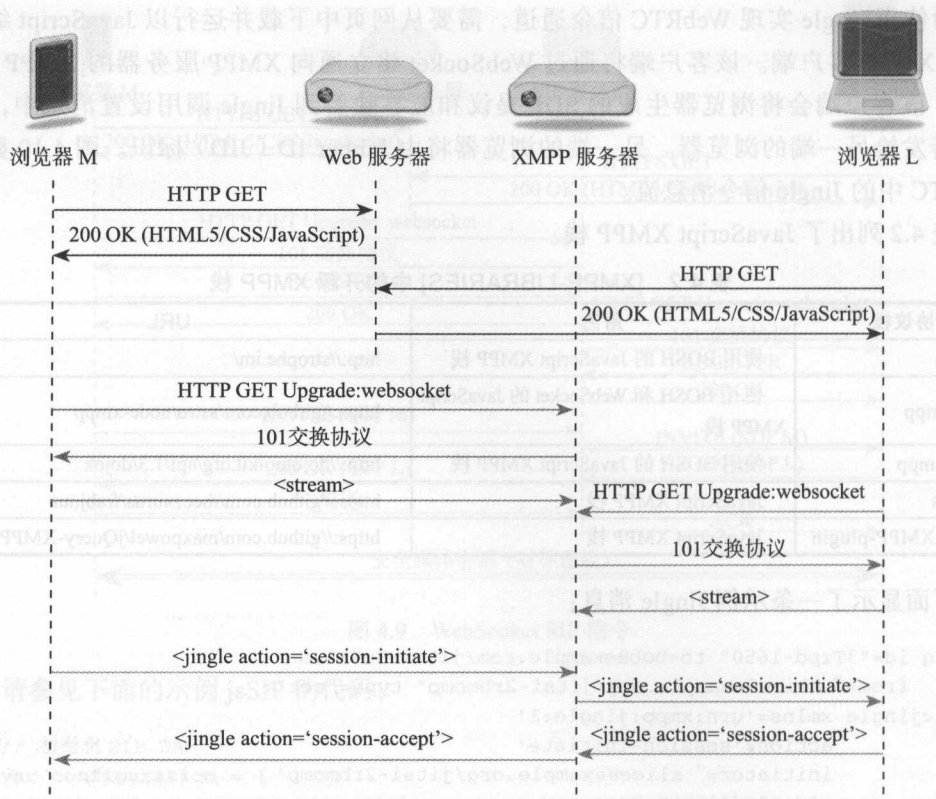


图 4.10 通过 WebSocket Jingle 传输 WebRTC 信令

4.3.8 数据通道专有信令

图 4.11 显示了数据通道专有信令。对于通过数据通道发送的信令消息，其性质可以完全是专有的。某些方案甚至不通过数据通道发送 SDP 对象，而是通过数据通道发送信令消息，再使用这些消息在本地生成要传递给浏览器的 SDP 对象。

4.3.9 使用叠加网络的数据通道

使用数据通道传输信令的另一种方案是使用数据通道构建一个叠加网络，并使用该叠加网络作为信令通道。这是一种对等（Peer-to-Peer，P2P）模式，有助于最大限度地减少对任何服务器类型的需求。

顾名思义，叠加网络就是叠加于或位于另一网络之上的网络。叠加网络隐藏了底层拓扑和体系结构，并为叠加网络中的其他成员提供了另一种寻址和消息传递方式。例如，环形网络就是一种叠加网络。叠加网络中的每个成员都跟踪处于环形中的其他相邻节点。举例而言，分布式散列表（Distributed Hash Table，DHT）是一种用于实现和利用叠加网络的技术。DHT 提供了一种将信息映射到叠加网络的方式。存在许多叠加网络路由协议，例

如 Chord [CHORD]，该协议规定了如何在叠加网络中存储和检索信息以及如何路由消息。Internet 工程任务组（Internet Engineering Task Force，IETF）的 P2PSIP 工作组 [PSPSIP-WG] 现已制定一些叠加网络标准，例如 RELOAD [RELOAD]。

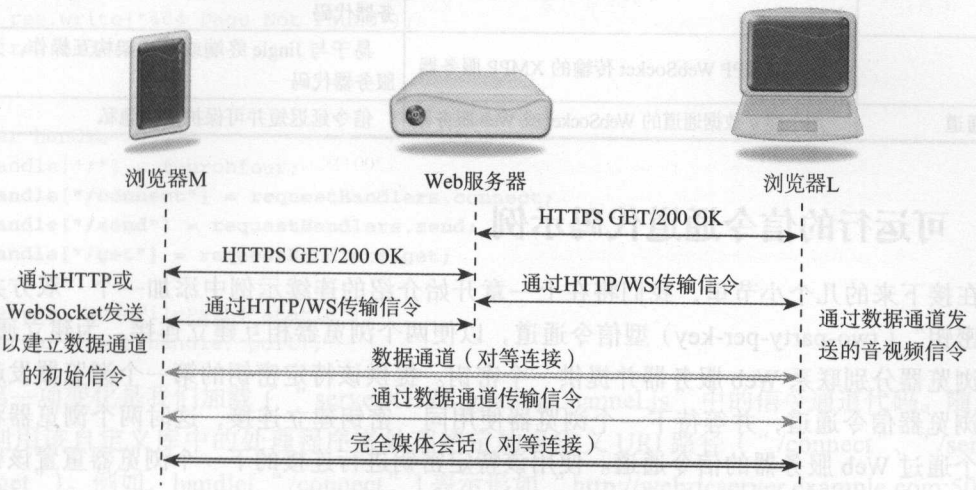


图 4.11 数据通道专有信令

数据通道可用于为 WebRTC 建立叠加信令网络。叠加网络协议将以 JavaScript 编写，并下载到浏览器中。Web 服务器充当引导服务器，并协助浏览器加入叠加网络，从而与叠加网络中的其他浏览器建立一些数据通道连接。一旦浏览器加入叠加网络，其与叠加网络之间的所有后续消息和信令都通过数据通道传输。只有当浏览器断开叠加网络连接并需要再次引导时，Web 服务器才会再次参与进来。

建立叠加网络后，叠加网络中的任何成员都可以与任何其他成员交换信令消息，进而建立对等连接。例如，可以使用 RELOAD 协议或开放对等端 [OPEN-PEER]。

对等信令网络的主要优点在于，它能够自我组织、自我扩展并保护隐私，对服务器的要求极少。随着叠加网络不断扩大，将在新成员之间分担附加信令负载。对等信令网络也具有普通对等网络的缺点。P2P 系统会占用成员的资源，因此需要更大的带宽和处理能力。此外，P2P 系统还必须应对可能试图打断叠加网络运行的恶意成员。

4.4 信令选项总结

表 4.3 总结了各种 WebRTC 信令方案。

表 4.3 WebRTC 信令选择对比

方案	服务器要求	优点
WebSocket 代理	提供服务器代码的 WebSocket 服务器	无需信令基础架构

（续）

方案	服务器要求	优点
XML HTTP 请求	提供服务器代码的 Web 服务器	无需信令基础架构
SIP	支持 SIP WebSocket 传输的 SIP 注册 / 代理服务器	易于与 SIP 终端或基础架构互操作，无需服务器代码
Jingle	支持 XMPP WebSocket 传输的 XMPP 服务器	易于与 Jingle 终端或基础架构互操作，无需服务器代码
数据通道	用于建立数据通道的 WebSocket 或 Web 服务器	信令延迟短并可保护信令隐私

4.5 可运行的信令通道代码示例

在接下来的几个小节里，我们将在上一章开始介绍的连续示例中添加一个“双方共享一个密钥”（two-party-per-key）型信令通道，以便两个浏览器相互建立连接。为建立通话，每个浏览器分别联系 Web 服务器并提供一个密钥。提供该特定密钥的第一个浏览器发起新的双浏览器信令通道，并等待下一个浏览器使用同一密钥建立连接，这时两个浏览器将拥有一个通过 Web 服务器的信令通道。使用该特定密钥进行连接的下一个浏览器重置该密钥的信令通道，然后重新等待下一个浏览器使用该密钥进行连接。

接下来的三节内容将介绍 Web 服务器代码、信令通道代码和 WebRTC 应用程序。请注意，前两者仍为通用代码，与 WebRTC API 无关。

4.5.1 Web 服务器

除了服务于 Web 应用程序本身之外，Web 服务器还可充当两个或更多浏览器之间的中继（提供“信令通道”），用于协商要在浏览器之间传输的媒体的目标、类型和格式。4.3.3 节和图 4.7 介绍了此处使用的 HTTP 轮询信令通道。

如前所述，WebRTC 应用程序是实时应用程序，因此需要一种快速高效的信令机制。在服务器代码中，我们将看到 Node 的共享内存空间如何在来自不同浏览器的独立浏览器请求之间简化通信。

下面我们来看看 index.js 中有哪些变化（更改的文本以粗体显示）。

4.5.1.1 index.js

```
// 版权所有 2013-2014 Digital Codex LLC 你可以将此代码用于自身学习
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责

var server = require("./server");
var requestHandlers = require("./serverXHRSignalingChannel");
var log = require("./log").log;
var port = process.argv[2] || 5001;
```

```
// 返回 404
function fourohfour(info) {
  var res = info.res;
  log("Request handler fourohfour was called.");
  res.writeHead(404, {"Content-Type": "text/plain"});
  res.write("404 Page Not Found");
  res.end();
}

var handle = {};
handle["/"] = fourohfour;
handle["/connect"] = requestHandlers.connect;
handle["/send"] = requestHandlers.send;
handle["/get"] = requestHandlers.get;

server.serveFilePath("static");
server.start(handle, port);
```

第一项变化是我们加载了“serverXHRSignalingChannel.js”中的信令通道代码。随后，我们利用该自定义库中的处理程序来处理特定的自定义 URI 路径（“/connect”、“/send”和“/get”）。例如，handle[“/connect”] 表示形如“http://webrtcserver.example.com:5001/connect”的 URI 将通过 serverXHRSignalingChannel.js 中定义的 connect() 处理程序处理。在介绍如何定义和使用这些处理程序来建立信令通道之前，我们先来快速看一下此阶段所需的一些服务器代码变更。

4.5.1.2 server.js

```
// 版权所有 2013-2014 Digital Codex LLC 你可以将此代码用于自身学习
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责

var http = require("http");
var url = require("url");
var fs = require('fs');

var log = require("./log").log;
var serveFileDir = "";

// 设置静态文件（HTML、JS 等）的路径
function setServeFilePath(p) {
  serveFilePath = p;
}
exports.serveFilePath = setServeFilePath;

// 创建一个处理程序来收集通过 POST 传输的数据并基于路径名称路由请求
function start(handle, port) {
```

```
  function onRequest(req, res) {
    var urldata = url.parse(req.url,true),
        pathname = urldata.pathname,
        info = {"res": res,
                "query": urldata.query,
                "postData":""};

    log("Request for " + pathname + " received");
    req.setEncoding("utf8");
    req.addListener("data", function(postDataChunk) {
      info.postData += postDataChunk;
      log("Received POST data chunk '"+ postDataChunk + "'.");
    });
    req.addListener("end", function() {
      route(handle, pathname, info);
    });
  }

  http.createServer(onRequest).listen(port);

  log("Server started on port " + port);
}

exports.start = start;


// 确定请求的路径是静态文件路径，还是拥有自己的处理程序的自定义路径
function route(handle, pathname, info) {
  log("About to route a request for " + pathname);
  // 检查前导斜杠后的路径是否为可处理的现有文件
  var filepath = createFilePath(pathname);
  log("Attempting to locate " + filepath);
  fs.stat(filepath, function(err, stats) {
    if (!err && stats.isFile()) {  // 处理文件
      serveFile(filepath, info);
    } else {  // 必须为自定义路径
      handleCustom(handle, pathname, info);
    }
  });
}

// 此函数先从给定路径名称中删除 ..、~和其他从安全角度而言存在问题的语法位，再向其开头添加 serveFilePath
// ** 我们并没有说此代码现已安全无虞 **
function createFilePath(pathname) {
  var components = pathname.substr(1).split('/');
  var filtered = new Array(),
      temp;

  for(var i=0, len = components.length; i < len; i++) {
    temp = components[i];
    if (temp == "..") continue;     // no updir
```

```
    if (temp == "") continue;          // no root
    temp = temp.replace(/~/g,'');      // no userdir
    filtered.push(temp);
  }
  return (serveFilePath + "/" + filtered.join("/"));
}

// 打开指定文件、读取其中的内容并将这些内容发送至客户端
function serveFile(filepath, info) {
  var res = info.res,
      query = info.query;

  log("Serving file " + filepath);
  fs.open(filepath, 'r', function(err, fd) {
    if (err) {log(err.message);
              noHandlerErr(filepath, res);
              return;}
    var readBuffer = new Buffer(20480);
    fs.read(fd, readBuffer, 0, 20480, 0,
      function(err, readBytes) {
        if (err) {log(err.message);
                  fs.close(fd);
                  noHandlerErr(filepath, res);
                  return;}
        log('just read ' + readBytes + ' bytes');
        if (readBytes > 0) {
          res.writeHead(200,
                        {"Content-Type": contentType(filepath)});
          res.write(
            addQuery(readBuffer.toString('utf8', 0, readBytes),
                     query));
        }
        res.end();
      });
  });
}

// 确定所提取的文件的内容类型
function contentType(filepath) {
  var index = filepath.lastIndexOf('.');

  if (index >= 0) {
    switch (filepath.substr(index+1)) {
      case "html":  return ("text/html");
      case "js":  return ("application/javascript");
      case "css":  return ("text/css");
      case "txt":  return ("text/plain");
      default:  return ("text/html");
    }
```

```
  }
  return ("text/html");
}

// 此函数设计用于HTML文件，可将文件中的第一个空脚本块替换为一个特定的对象，该对象表示请求URI
中包含的所有查询参数。
function addQuery(str, q) {
  if (q) {
    return str.replace('<script></script>',
                       '<script>var queryparams = ' +
                         JSON.stringify(q) + ';</script>');
  } else {
    return str;
  }
}

// 确认非文件路径的处理程序，然后执行该程序
function handleCustom(handle, pathname, info) {
  if (typeof handle[pathname] == 'function') {
    handle[pathname](info);
  } else {
    noHandlerErr(pathname, info.res);
  }
}

// 如果没有为请求定义处理程序，则返回404
function noHandlerErr(pathname, res) {
  log("No request handler found for " + pathname);
  res.writeHead(404, {"Content-Type": "text/plain"});
  res.write("404 Page Not Found");
  res.end();
}
```

在此阶段添加了两项主要功能：一是服务器能够接受通过 POST 发送的数据；二是使用 URL 查询参数初始化特定变量的机制。POST 数据由 onRequest() 处理程序寻址，方法是设置处理程序，以在通过 POST 发送的数据传入时收集它们，并只在所有 POST 数据都收集完毕后才调用 route()。

经过分析后，查询参数将保存在 start() 函数的标头中，并通过 info 变量设置为可用，但由 addQuery() 进行处理。

这里，addQuery() 要了一个小把戏，可在内容被返回之前以动态方式将其插入（HTML）文件中。此特定函数设计用于提取表示请求 URI 中给定查询参数的对象，并将其插入文件中。在此 WebRTC 示例中，通过这一函数，可以将请求 URI 中指定为查询参数的密钥插入所提取的文件，以供浏览器使用。查询参数将被转换为 JSON 字符串，并插入文件中的第

一个空脚本块。不要认为此代码示例是安全的。只要用户输入最终能够生成一个文件，就存在遭到恶意人士滥用的风险，因此在向自己的代码中添加任何此类功能之前，一定要弄清楚这样做的后果。

4.5.2 信令通道

WebRTC 允许浏览器使用任意通信方法来传输信令信息，从烟雾信号到带字符串的 Tin Can，再到更为快捷且更常用的方案，比如 HTTP（XML HTTP 请求）、WebSocket 和 Google 应用程序引擎通道，不一而足。本章前面已经探讨并比较了这些不同的方案。但请注意，无论在何种情况下，浏览器都是与 Web 服务器进行通信，而非另一端的浏览器。虽然不限制浏览器彼此直接通信，但 Web 模型本身假定，浏览器（Web 客户端）只与服务器通信，而不直接与其他 Web 客户端通信。实际上，除非 Web 服务器提供了 Web 客户端的 IP 地址，否则这些客户端根本不知道如何相互联系。因此，在大多数情况下，云中都有一台服务器充当中继，用于将消息从一个浏览器发送至另一个浏览器。

一般而言，基于 Web 服务器的信令方案都属于以下两个类别之一：轮询方案和面向会话的方案。在以 XHR 为代表的轮询方案中，所有通信都由浏览器发起，浏览器从 Web 服务器接收消息的唯一方式就是接收服务器对其消息的响应。为了接收来自服务器的全部消息，浏览器需要定期向服务器发送消息来进行轮询，以防存在等待接收的消息。在面向会话的方案（例如 WebSocket）中，将在浏览器和 Web 服务器之间建立一个虚拟连接，以便相互发送消息。

为简化应用程序编程操作，一种便捷的做法是定义一个信令通道接口，用于公开 connect 和 send 方法，并能够针对消息的接收时间指定处理程序。如果应用程序要在对等模式（“三角形”模式）中使用 WebRTC，则该虚拟信令通道的作用是提供一个简单的面向连接的模型，用于向对等端发送信令信息。创建此虚拟连接时，有几个方面的问题需要注意。第一个问题是确定如何向 Web 服务器指示要建立的连接的性质，这个问题最为重要。此连接真的只是面向一个对等端，还是要建立群组“通话”？假定它是一对一连接，如何对另一方进行标识？一种合理的解决方案是让每个浏览器都在服务器中注册，并让服务器发回已注册浏览器的列表，以便用户选择要与哪个浏览器进行通信。另一种解决方案是使用另一端的浏览器知晓并提供的密钥。本章中的演示代码使用的就是后一种方案。在此特定演示中，使用给定密钥进行连接的第一个浏览器注册为第一方，并被告知进行等待。当第二个客户端使用同一密钥进行连接时，将在两个客户端之间建立信令通道。随后，如果有新的一方尝试使用同一密钥建立另一个连接，将销毁现有连接，并指示其进行等待。接下来，当另一个浏览器使用同一密钥进行连接时，将在这两者之间建立信令通道，以此类推，其状态就在 waiting 与 connected 之间变换。此方案的潜在缺点在于，双方都需要知道其密钥，并且需要由新的一方来销毁连接。尽管如此，它也有两个非常有利的特性：密钥可作为参数包含在 URI 中，这样只需将该 URI 添加为书签，即可保存该“连接”，而且当连接因为

浏览器或网络故障而断开时，只需管理极少的状态。当连接断开时，双方只需重新加载其页面，即可重新开始通信。

下面，我们根据上面的介绍来看一下代码。本演示中的信令通道代码分拆在两个文件中，其中一个文件位于服务器上，另一个位于客户端上。我们先来看一下serverXHRSignalingChannel.js 中的服务器代码。

4.5.2.1 serverXHRSignalingChannel.js

```
// 版权所有 2013-2014 Digital Codex LLC 你可以将此代码用于自身学习
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责

var log = require("./log").log;

var connections = {},
    partner = {},
    messagesFor = {};


// 排队发送 JSON 响应
function webrtcResponse(response, res) {
  log("replying with webrtc response " +
      JSON.stringify(response));
  res.writeHead(200, {"Content-Type":"application/json"});
  res.write(JSON.stringify(response));
  res.end();
}


// 发送错误作为 JSON WebRTC 响应
function webrtcError(err, res) {
  log("replying with webrtc error:  " + err);
  webrtcResponse({"err": err}, res);
}


// 处理 XML HTTP 请求以使用给定密钥进行连接
function connect(info) {
  var res = info.res,
      query = info.query,
      thisconnection,
      newID = function() {
        // 创建一个大的随机数字，此数字应不可能在服务器生命周期中很快重复出现
        return Math.floor(Math.random()*1000000000);
      },
      connectFirstParty = function() {
        if (thisconnection.status == "connected") {
          // 删除配对和任何存储的消息
```

```
        delete partner[thisconnection.ids[0]];
        delete partner[thisconnection.ids[1]];
        delete messagesFor[thisconnection.ids[0]];
        delete messagesFor[thisconnection.ids[1]];
      }
      connections[query.key] = {};
      thisconnection = connections[query.key];
      thisconnection.status = "waiting";
      thisconnection.ids = [newID()];
      webrtcResponse({"id":thisconnection.ids[0],
                      "status":thisconnection.status}, res);
    },
    connectSecondParty = function() {
      thisconnection.ids[1] = newID();
      partner[thisconnection.ids[0]] = thisconnection.ids[1];
      partner[thisconnection.ids[1]] = thisconnection.ids[0];
      messagesFor[thisconnection.ids[0]] = [];
      messagesFor[thisconnection.ids[1]] = [];
      thisconnection.status = "connected";
      webrtcResponse({"id":thisconnection.ids[1],
                      "status":thisconnection.status}, res);
    };

  log("Request handler 'connect' was called.");
  if (query && query.key) {
    var thisconnection = connections[query.key] ||
                         {status:"new"};
    if (thisconnection.status == "waiting") { // 前半部分就绪
      connectSecondParty(); return;
    } else { // 必须为新连接或“connected”状态
      connectFirstParty(); return;
    }
  } else {
    webrtcError("No recognizable query key", res);
  }
}
exports.connect = connect;

// 对 info.postData.message 中的消息排队，以发送至具有
// info.postData.id 中的 ID 的伙伴
function sendMessage(info) {
  log("postData received is ***" + info.postData + "***");
  var postData = JSON.parse(info.postData),
      res = info.res;

  if (typeof postData === "undefined") {
    webrtcError("No posted data in JSON format!", res);
    return;
  }
```

```
  if (typeof (postData.message) === "undefined") {
    webrtcError("No message received", res);
    return;
  }
  if (typeof (postData.id) === "undefined") {
    webrtcError("No id received with message", res);
    return;
  }
  if (typeof (partner[postData.id]) === "undefined") {
    webrtcError("Invalid id " + postData.id, res);
    return;
  }
  if (typeof (messagesFor[partner[postData.id]]) ===
              "undefined") {
    webrtcError("Invalid id " + postData.id, res);
    return;
  }
  messagesFor[partner[postData.id]].push(postData.message);
  log("Saving message ***" + postData.message +
      "*** for delivery to id " + partner[postData.id]);
  webrtcResponse("Saving message ***" + postData.message +
                 "*** for delivery to id " +
                 partner[postData.id], res);
}
exports.send = sendMessage;

// 返回所有排队获取 info.postData.id 的消息
function getMessages(info) {
  var postData = JSON.parse(info.postData),
      res = info.res;

  if (typeof postData === "undefined") {
    webrtcError("No posted data in JSON format!", res);
    return;
  }
  if (typeof (postData.id) === "undefined") {
    webrtcError("No id received on get", res);
    return;
  }
  if (typeof (messagesFor[postData.id]) === "undefined") {
    webrtcError("Invalid id " + postData.id, res);
    return;
  }

  log("Sending messages ***" +
      JSON.stringify(messagesFor[postData.id]) + "*** to id " +
      postData.id);
  webrtcResponse({'msgs':messagesFor[postData.id]}, res);
```

```
    messagesFor[postData.id] = [];
  }
  exports.get = getMessages;
```

此代码首先要求记录框架并建立变量来承载以下对象：1）连接列表；2）用于在对等端之间进行映射的简单数组；3）包含要发送给客户的消息的数组。在定义代码的主体（定义如何连接、如何发送消息和如何检索消息）之前，我们先定义两个频繁使用的函数，来处理针对 HTTP 请求的答复：一是 webrtcResponse()，用于将任何传入的对象转换为 JSON 字符串并返回该字符串；二是 webrtcError()，用于通过 webrtcResponse() 以“ err”属性值的形式返回给定的对象（可能是字符串）。

三个主要处理程序为 connect()、sendMessage() 和 getMessages()。除了提取文件之外，connect() 是浏览器与 WebRTC 服务器进行的第一项交互。原始请求及所有查询参数都将作为 info 对象的属性传入。要理解此代码，最简便的方式是跳过此函数的所有定义，直接查看此函数的末尾。在记录了 connect 处理程序的调用事件后，此代码确认该 URI 包含密钥，例如“ http://webrtc.example.org:5001/connect&key=1234”。如果 connections[] 数组中已存在相应的连接，则找到该连接，否则将创建一个新连接。随后，如果连接为新连接或已建立的连接，此代码将调用 connectFirstParty()；如果连接的状态为 waiting，则调用 connectSecondParty()。现在我们来看一下 connectFirstParty()，你将会发现，我们执行的第一项操作是删除使用该密钥的任何现有连接，以及 partner[] 和 messagesFor[] 数组中相应的条目；关于这两个数组，我们将在后面介绍。没错，如果此操作发生在已连接的实时通话期间，这将删除该通话的“信令通道”，阻止传输任何其他信令。当然，如果有任何媒体直接在双方之间流动，它将继续流动，因为这正是对等媒体的含义！我们可以让代码检查信令通道是否中断，并在中断时采取有效的操作，例如重试；不过，本代码示例并没有这样处理。我们继续看这段代码，在删除使用该密钥的各个现有连接后，将为该密钥创建一个新的 connections[] 条目，并将其状态设置为 waiting，同时创建一个新的 ids[] 数组属性，使其包含为此浏览器客户端生成的新 ID。最后，将针对连接请求生成响应，且其中包含新的 ID 和状态。

connectSecondParty() 行为与 connectFirstParty() 类似，但在此示例中，我们：

1）不删除现有（waiting）连接。

2）创建要添加到连接中的第二个 ID。

3）设置一个辅助数组，通过 ID 关联两个伙伴（对等端）并存储一方向另一方发送的消息。

这体现了使用的 Node 的一项关键优势——发送至服务器的所有请求都使用同一个共享内存空间，并且只存在一个控制线程，因此所有请求均可安全访问同一连接数组。很快我们将会看到，所有消息都单纯以值的形式存储在这些共享数组（位于单个内存空间）中。

sendMessage() 是形如“ http://webrtc.example.org:5001/send”的 URI 的处理程序。请注

意，我们假定来自客户端的所有信息都编码为 POST 数据。在完成各项错误检查后，包括检查是否缺少 POST 数据、没有消息、没有 ID 或缺少连接（这意味着 partner[] 或 messagesFor[] 数组中没有条目），我们将消息推送到与此客户端的伙伴相对应的 messagesFor 条目的末尾。随后，我们向客户端发回一条状态消息。

最后，getMessages() 是形如“ http://webrtc.example.org:5001/get”的 URI 的处理程序。与 sendMessage() 一样，我们假定来自客户端的所有信息都编码为 POST 数据。同样，对于此客户端 ID，我们要确保存在相应的 POST 数据、ID 和 messagesFor[] 数组条目，随后我们以 msgs 属性值的形式发回消息数组。此外，我们还重置此 ID 的 messagesFor[] 条目，以免反复收到相同的消息。

由于采用 Node 和异步 JavaScript，关于何时发回响应，存在一个微妙之处。由于只有一个控制线程，因此只有在我们的代码运行完毕之后，才会发送响应。为明确起见，对于此代码中的每一条路径，此代码基本上将响应都设置为返回操作之前执行的最后一项操作。

现在我们来看一下客户端中的信令代码，此代码包含在 clientXHRSignalingChannel.js 中。

4.5.2.2 clientXHRSignalingChannel.js

```
// 版权所有 2013-2014 Digital Codex LLC 你可以将此代码用于自身学习
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世。使用此代码的风险完全由你自己承担。
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责。


// 此代码将创建客户端命令，用于建立基于 XML HTTP 请求的 WebRTC 信令通道。


// 此信令通道假定通过共享密钥建立双人连接。
// 每次连接尝试都会导致状态在“waiting”和“connected”之间切换，
// 这意味着如果两个浏览器已建立连接，
// 而另一个浏览器尝试进行连接，将断开现有连接，并且新浏览器的状态将变为“waiting”。


var createSignalingChannel = function(key, handlers) {


var id, status, doNothing = function(){},
  handlers = handlers || {},
  initHandler = function(h) {
    return ((typeof h === 'function') && h) || doNothing;
  },
  waitingHandler = initHandler(handlers.onWaiting),
  connectedHandler = initHandler(handlers.onConnected),
  messageHandler = initHandler(handlers.onMessage);


// 与信令服务器建立连接
```

```
function connect(failureCB) {
  var failureCB = (typeof failureCB === 'function') ||
                  function() {};

  // 处理连接响应，该响应应为错误或状态 (of "connected" 或 "waiting")
  function handler() {
    if(this.readyState == this.DONE) {
      if(this.status == 200 && this.response != null) {
        var res = JSON.parse(this.response);
        if (res.err) {
          failureCB("error:  " + res.err);
          return;
        }

        // 如果没有错误，则保存状态和服务器生成的 ID，
        // 然后启动异步消息轮询
        id = res.id;
        status = res.status;
        poll();

        // 运行用户提供的处理程序来处理 waiting 和 connected 状态
        if (status === "waiting") {
          waitingHandler();
        } else {
      connectedHandler();
        }
        return;
      } else {
        failureCB("HTTP error:  " + this.status);
        return;
      }
    }
  }

  // 打开 XHR 并发送包含密钥的连接请求
  var client = new XMLHttpRequest();
  client.onreadystatechange = handler;
  client.open("GET", "/connect?key=" + key);
  client.send();
}

// poll() 会在访问服务器之前等待 n 毫秒。
// 对于前 10 次尝试，n 为 10 毫秒；对于接下来的 10 次尝试，n 为 100 毫秒；对于后续尝试，n 均为 1000 毫秒。
// 如果实际收到消息，则将 n 重置为 10 毫秒。

function poll() {
  var msgs;
  var pollWaitDelay = (function() {
    var delay = 10, counter = 1;

    function reset() {
```

```
    delay = 10;
    counter = 1;
  }

  function increase() {
    counter += 1;
    if (counter > 20) {
      delay = 1000;
    } else if (counter > 10) {
      delay = 100;
    }                          // else leave delay at 10
  }

  function value() {
    return delay;
  }

  return {reset: reset, increase: increase, value: value};
}());

// 此处立即定义并使用了 getLoop。
// 它从服务器中检索消息，然后将自身计划为在 pollWaitDelay.value() 毫秒后重新运行。

(function getLoop() {
  get(function (response) {
    var i, msgs = (response && response.msgs) || [];

    // 如果存在消息属性，则表示我们已建立连接
    if (response.msgs && (status !== "connected")) {
      // 将状态切换为 connected，因为现在确实已建立连接！
      status = "connected";
      connectedHandler();
    }
    if (msgs.length > 0) {             // 我们收到消息
      pollWaitDelay.reset();
      for (i=0; i<msgs.length; i+=1) {
        handleMessage(msgs[i]);
      }
    } else {                           // 没有收到任何消息
      pollWaitDelay.increase();
    }

    // 现在设置计时器以便重新检查
    setTimeout(getLoop, pollWaitDelay.value());
  });
}());
}

// 此函数是轮询设置的一部分，该设置用于检查是否有来自另一端浏览器的消息。
```

```
// 它由 poll() 中的 getLoop() 调用。
function get(getResponseHandler) {

  // 响应应为错误或 JSON 对象。
  // 如果是后者，则将其发送给用户提供的处理程序。
  function handler() {
    if(this.readyState == this.DONE) {
      if(this.status == 200 && this.response != null) {
        var res = JSON.parse(this.response);
        if (res.err) {
          getResponseHandler("error:  " + res.err);
          return;
        }
        getResponseHandler(res);
        return res;
      } else {
        getResponseHandler("HTTP error:  " + this.status);
        return;
      }
    }
  }

  // 打开 XHR 并针对我的 ID 请求消息
  var client = new XMLHttpRequest();
  client.onreadystatechange = handler;
  client.open("POST", "/get");
  client.send(JSON.stringify({"id":id}));
}

// 计划传入的消息以进行异步处理。
// 此函数由 poll() 中的 getLoop() 使用。
function handleMessage(msg) {    // 异步处理消息
  setTimeout(function () {messageHandler(msg);}, 0);
}

// 通过信令通道向另一端的浏览器发送消息
function send(msg, responseHandler) {
  var reponseHandler = responseHandler || function() {};

  // 分析响应并发送给处理程序
  function handler() {
    if(this.readyState == this.DONE) {
      if(this.status == 200 && this.response != null) {
        var res = JSON.parse(this.response);
        if (res.err) {
          responseHandler("error:  " + res.err);
          return;
```

```
        }
        responseHandler(res);
        return;
      } else {
        responseHandler("HTTP error:  " + this.status);
        return;
      }
    }
  }

  // 打开 XHR 并以 JSON 字符串形式发送我的 ID 和消息
  var client = new XMLHttpRequest();
  client.onreadystatechange = handler;
  client.open("POST", "/send");
  var sendData = {"id":id, "message":msg};
  client.send(JSON.stringify(sendData));
}


return {
  connect:  connect,
  send:  send
};

};
```

createSignalingChannel() 接收共享密钥和一组处理程序，并返回（在文件末尾）一个包含 connect() 和 send() 方法的对象。此代码首先初始化一些变量，包括用于处理 waiting 和 connected 状态以及消息接收事件的处理程序。

connect() 带有回调，旨在免连接请求出现问题。我们先来看一下这个例程的末尾。对于连接请求，将创建一个新的 XMLHttpRequest()。设置处理程序后（稍后讨论），向 Web 服务器上的“/connect”URI 发送 GET（相对于 POST 而言）请求，并以查询参数的形式附上密钥。对于该处理程序，首先要注意的一点是，XMLHttpRequest() 中有多种事件可触发该处理程序。在此代码中，我们唯一关注的情形是结果完整且经过分析，即结果的 readyState 为 DONE。随后，此代码确认存在成功的 HTTP 响应（状态为 200），否则将调用 failureCB()。从该处理程序的下一行代码可以看出一个有趣的情况，即 XMLHttpRequest() 实际上可以接收任何格式的任意数据，而不局限于 XML。在该示例中，我们将响应作为 JSON 进行分析，并先检查返回的对象是否包含错误响应。如果包含，我们使用该错误调用 failureCB() 回调。否则，我们将分析出服务器为此浏览器客户端生成的 ID 及其状态（waiting 或 connected），然后开始异步消息轮询（稍后讨论）。随后，我们针对返回的状态调用相应的处理程序，以此来结束响应处理。

在本文件的代码中，poll() 是最复杂的部分，其唯一的作用就是频繁检查是否有来自对等端的消息（通过服务器）。如果采用的信令通道方案不是面向会话的方案，则有必要使用

此类代码。此例程的第一部分定义了一个计数器对象 pollWaitDelay。如果调用 reset()，则将该对象的值设置为 10，否则每次调用 increase() 都将增加该计数器的值。如果重置后的计数达到 11，该值将变为 100；如果重置后的计数达到 21，该值将变为 1000。此值将用作发出下一个 get 请求之前等待的毫秒数。这种设计的目的在于，在收到一些消息后快速检查新的消息，如果没有新消息到来，则将检查频率恢复为较低的水平，这样处理是为了在收到消息时快速响应，并在不交换消息时避免给服务器带来繁重的负载。

getLoop() 是计划 get 请求的例程。它调用 get()，此函数从服务器请求任何正在等待的消息，并将服务器的响应作为参数传递给指定的处理程序。该处理程序在 getLoop() 中以内嵌方式定义，它会先创建一个 msgs 数组，再向其中填充服务器发来的消息（如果有）。只有当客户端浏览器之间已建立连接时，服务器才会发回具有 msgs 属性的响应，因此如果存在此类属性，而浏览器尚不知道已建立连接，则现在可将其标记为 connected 并运行 connectedHandler()。如果浏览器是进行连接的第一方，即它在等待另一浏览器进行连接，就会出现这种情况。如果收到具有 msgs 属性的响应，则表示已完成连接，即使实际中没有消息，也不例外。接下来的情形取决于是否返回了任何消息。如果返回了消息，将出现两种情况：一是重置时间计数器；二是将消息发送给消息处理程序以进行异步处理。重置时间计数器是为了确保另一端的浏览器紧随之前消息发来的任何消息都能被快速接收。无论在哪种情况下，getLoop() 都计划为在 pollWaitDelay.value() 毫秒后重新运行。

get() 实际上执行消息提取操作，方法是将响应作为输入发送给指定的处理程序。此代码的主要部分位于例程末尾，其作用十分类似于 connect()——向服务器发送新的 XMLHttpRequest。但在此示例中，URI 指向“/get”，我们还同时发送此客户端浏览器的 ID（在 connect() 调用中分配给浏览器），该 ID 编码为 JSON 字符串。与 connect() 一样，响应处理程序将会：

1）确认 readyState 为 DONE（表示它为最终响应）。

2）确认 HTTP 请求已成功（状态将为 200 且响应非空）。

3）确认响应不包含 err 属性。

如果一切顺利，则会将整个响应发送给作为参数传入 get() 方法的响应处理程序。

下一段代码是 handleMessage()，对于收到的每一条消息，都将从 poll() 中的 getLoop() 内调用此代码。其作用是计划要对消息异步运行的信令通道 messageHandler() 回调。

最后一段客户端信令代码是 send() 的定义，此例程向对等端发送有关信令通道的消息。主体代码位于此方法末尾，其作用十分类似于 connect() 和 get()——向服务器发送新的 XMLHttpRequest。但在此示例中，URI 指向“/send”，我们还同时发送此客户端浏览器的 ID（在 connect() 调用中分配给浏览器）和消息（以参数形式提供给 send()）。同样，这二者也编码为 JSON 字符串。同时，与 connect() 和 get() 一样，例程的响应处理程序将会：

1）确认 readyState 为 DONE（表示此为最终响应）。

2）确认 HTTP 请求已成功（状态将为 200 且响应非空）。

3）确认响应不包含 err 属性。

与 get() 一样，我们只将整个响应传递给作为参数传入 send() 方法的响应处理程序。请注意，与 connect() 和 get() 不同的是，对于 send()，我们并不期望响应中包含任何实际信息，它主要用于传递错误，也可能用于传递可记录的状态消息。

4.5.3 客户端 WebRTC 应用程序

我们在现有演示的基础上再次加入代码，这次添加的是信令通道。我们先来看代码，然后再加以解释。

```
<!--
// 版权所有 2013-2014 Digital Codex LLC
// 你可以将此代码用于自身学习。
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，请勿标榜你的代码先于此代码问世。
// 使用此代码的风险完全由你自己承担。
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责。
-->

<html>
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
  <style>
    video {
      width:  320px;
      height:  240px;
      border:  1px solid black;
    }
    div {
      display:  inline-block;
    }
  </style>
</head>
<body>

<!-- 此空白脚本部分专为查询参数预留 -->
<script></script>

<!--  加载 polyfill，先加载本地副本以进行本地测试 -->
<script src="extra/adapter.js" type="text/javascript"></script>
<script
    src="https://webrtc.github.io/adapter/adapter.js"
    type="text/javascript"></script>

<!--  加载基于 XHR 的信令通道，以基于密钥定向连接 -->
<script src="clientXHRSignalingChannel.js"
        type="text/javascript"></script>

<script>
```

```
var signalingChannel, key, id,
    haveLocalMedia = false,
    connected = false,
    myVideoStream, myVideo;

////////////////////////////
// 这是主例程。
////////////////////////////

// 由此开始获取本地媒体。此外，它还可以自动启动信令通道。
window.onload = function () {

  // 如果URI中提供了密钥，则自动连接信令通道
  if (queryparams && queryparams['key']) {
    document.getElementById("key").value = queryparams['key'];
    connect();
  }

  myVideo = document.getElementById("myVideo");

  getMedia();
};


//////////////////////
// 接下来这一节代码用于建立信令通道。
//////////////////////

// 此例程会连接至Web服务器并建立信令通道。
// 当出现文档负载或用户单击“Connect”(连接)按钮时，将自动调用此例程。
function connect() {
  var errorCB, scHandlers, handleMsg;

  // 首先，获取用于连接的密钥
  key = document.getElementById("key").value;

  // 此处理程序用于处理通过信令通道收到的所有消息
  handleMsg = function (msg) {
    // 将消息发布到屏幕上
    var msgE = document.getElementById("inmessages");
    var msgString = JSON.stringify(msg);
    msgE.value = msgString + "\n" + msgE.value;
  };

  // 用于信令通道的处理程序
  scHandlers = {
    'onWaiting' : function () {
      setStatus("Waiting");
    },
    'onConnected': function () {
```

```
      connected = true;
      setStatus("Connected");
      // 等待本地媒体准备就绪
      verifySetupDone();
    },
    'onMessage': handleMsg
  };

  // 最后，创建信令通道
  signalingChannel = createSignalingChannel(key, scHandlers);
  errorCB = function (msg) {
    document.getElementById("response").innerHTML = msg;
  };

  // 进行连接
  signalingChannel.connect(errorCB);
}

// 此例程通过信令通道发送消息，其方式有两种：一是执行显式调用；二是通过用户单击“Send”(发送)按钮。
function send(msg) {
  var handler = function (res) {
    document.getElementById("response").innerHTML = res;
    return;
  },

  // 如果没有传入，则获取消息
  msg = msg || document.getElementById("message").value;

  // 发布到屏幕上
  msgE = document.getElementById("outmessages");
  var msgString = JSON.stringify(msg);
  msgE.value = msgString + "\n" + msgE.value;

  // 并通过信令通道发送
  signalingChannel.send(msg, handler);
}

////////////////////////////
// 接下来的这一节代码用于获取本地媒体
////////////////////////////

function getMedia() {
  getUserMedia({"audio":true, "video":true},
               gotUserMedia, didntGetUserMedia);
}

function gotUserMedia(stream) {
  myVideoStream = stream;
```

```
  haveLocalMedia = true;

  // 向我显示我的本地视频
  attachMediaStream(myVideo, myVideoStream);
  // 等待建立信令通道
  verifySetupDone();
}

function didntGetUserMedia() {
  console.log("couldn't get video");
}

// 此守护例程实际上用于对两项异步活动
// 的完成时间进行同步：一是创建信令通道；
// 二是获取本地媒体。
function verifySetupDone() {
  // 如果信令通道已经就绪，并且我们已获得本地媒体，则继续处理。
  if (connected && haveLocalMedia) {setStatus('Set up');}
}

////////////////////////////////////
// 这一节代码用来基于应用程序的进度更改 UI。
////////////////////////////////////

// 此函数将通过隐藏、显示和填充各种 UI 元素，
// 让用户大体了解浏览器在建立信令通道和获取本地媒体方面的进度。
function setStatus(str) {
  var statuslineE = document.getElementById("statusline"),
      statusE = document.getElementById("status"),
      sendE = document.getElementById("send"),
      connectE = document.getElementById("connect"),
      scMessageE = document.getElementById("scMessage");

  switch (str) {
    case 'Waiting':
      statuslineE.style.display = "inline";
      statusE.innerHTML =
        "Waiting for peer signaling connection";
      sendE.style.display = "none";
      break;
    case 'Connected':
      statuslineE.style.display = "inline";
      statusE.innerHTML =
        "Peer signaling connected, waiting for local media";
      sendE.style.display = "inline";
      scMessageE.style.display = "inline-block";
      break;
    case 'Set up':
      statusE.innerHTML =
        "Peer signaling connected and local media obtained";
      break;
```

```
    default:
  }
}

</script>

<div id="setup">
  <p>WebRTC Book Demo (local media and signaling only)</p>
  <p>Key:
    <input type="text" name="key" id="key"
           onkeyup="if (event.keyCode == 13) {
                    connect(); return false;}"/>
    <button id="connect" onclick="connect()">Connect</button>
    <span id="statusline" style="display:none">Status:
      <span id="status">Disconnected</span>
    </span>
  </p>
</div>

<div id="scMessage" style="float:right;display:none">
  <p>Signaling channel message:
    <input type="text" width="100%" name="message" id="message"
           onkeyup="if (event.keyCode == 13) {
                    send(); return false;}"/>
    <button id="send" style="display:none"
            onclick="send()">Send</button>
  </p>

  <p>Response:  <span id="response"></span></p>
</div>

<br/>

<div style="width:30%;vertical-align:top">
  <div>
    <video id="myVideo" autoplay="autoplay" controls
           muted="true"/>
  </div>
  <p><b>Outgoing Messages</b>
     <br/>
     <textarea id="outmessages" rows="100"
               style="width:100%"></textarea>
  </p>
</div>

<div style="width:30%;vertical-align:top">
  <div>
    <video id="placeholder" autoplay="autoplay" controls />
  </div>
  <p><b>Incoming Messages</b>
     <br/>
```

```
        <textarea id="inmessages" rows="100"
                  style="width:100%"></textarea>
      </p>
    </div>

    </body>
    </html>
```

与之前一样，我们先从 HTML 标记所在的文件末尾开始讲解。图 4.12 显示了此应用程序在 Firefox 上的视觉效果。现在，这里有三个主要的区域：左上方的应用程序控制区、右上方的状态和信令消息区（默认通过“display:none”隐藏），以及下方的视频 / 消息区。

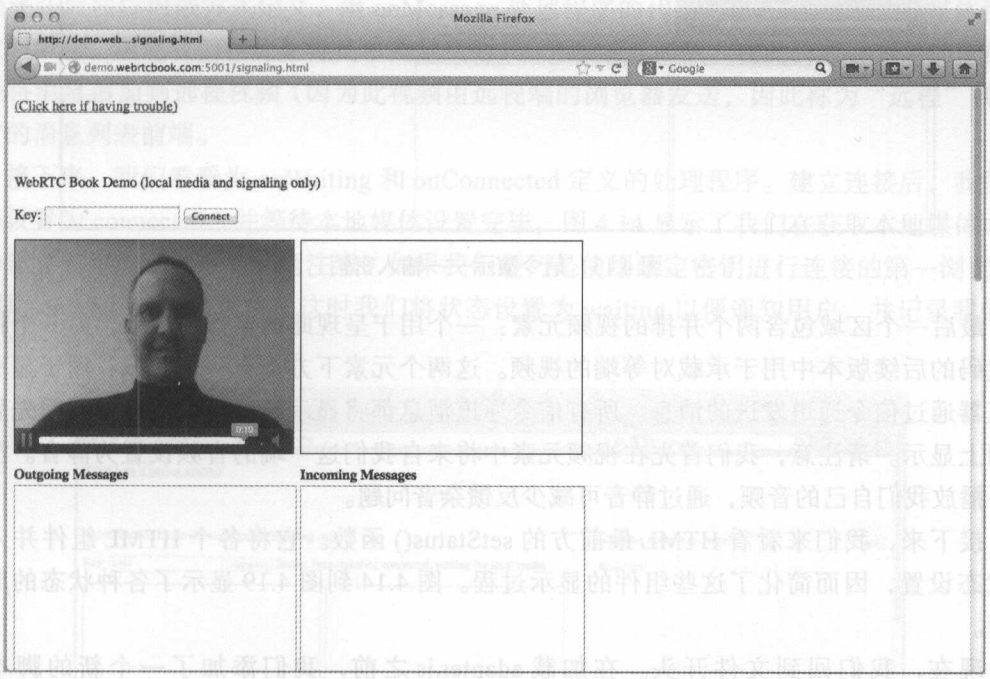


图 4.12 信令演示——连接之前

左上方的应用程序控制区最初会显示两项内容：一是用于输入密钥的字段；二是“Connect”（连接）按钮，单击该按钮后，将调用 connect() 函数，以使用该密钥联系服务器并建立信令通道。稍后我们将介绍相应的代码。此区域还包含一个状态行，其最初处于隐藏状态（“display:none”）。图 4.13 显示了已在字段中输入密钥但尚未单击“Connect”（连接）按钮时的情况。请注意，此时尚不能看到状态行。

对于右上方的状态和信令消息区，我们将在查看激活该区域的代码时进行讲解，现在只需知道一点，即它将包含一个信息区，其中含有各个 XMLHttpRequest 的响应，并能够通过信令通道手动发送消息。此区域主要用于测试目的，但也提供通过信令通道进行聊天的初步功能。在图 4.13 中，现在同样还看不到此区域。

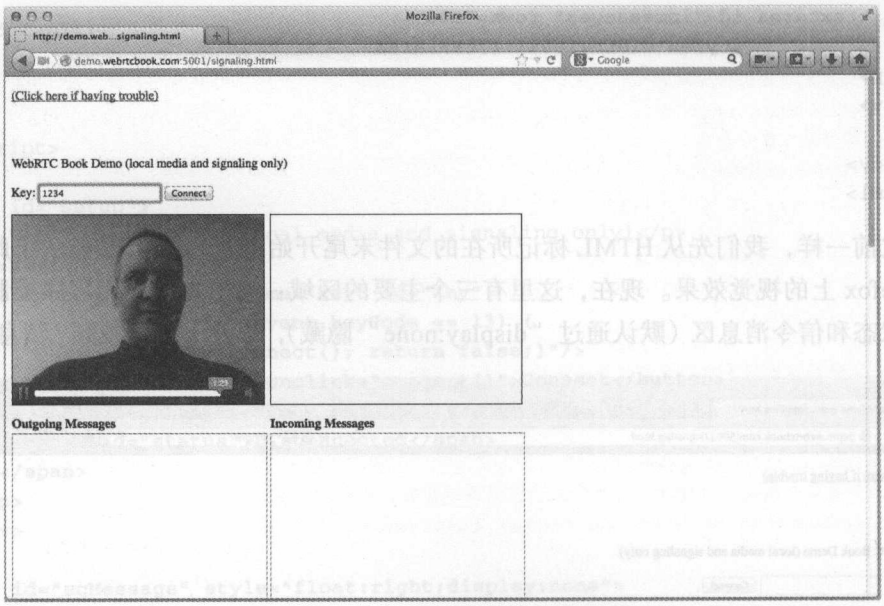


图 4.13 信令演示——输入密钥

最后一个区域包含两个并排的视频元素：一个用于呈现此浏览器的视频，另一个将在此代码的后续版本中用于承载对等端的视频。这两个元素下方都有一个区域，用于显示该浏览器通过信令通道发送的消息。所有信令通道消息都将显示在这里，消息按时间先后由下而上显示。请注意，我们首先在视频元素中将来自我们这一端的音频设置为静音。没有必要播放我们自己的音频，通过静音可减少反馈杂音问题。

接下来，我们来看看 HTML 最前方的 setStatus() 函数。它将各个 HTML 组件并入一项状态设置，因而简化了这些组件的显示过程。图 4.14 到图 4.19 显示了各种状态的屏幕截图。

现在，我们回到文件开头。在加载 adapter.js 之前，我们添加了一个新的脚本块 <script></script>。没错，此脚本块是空的！还记得我们的服务器代码吗？ server.js 中的 addQuery() 将在此脚本块中插入请求此文件的 URI 中包含的所有查询参数。这些参数将作为新的本地 queryparams 变量的属性插入。现在，我们还加载了刚刚介绍过的 clientXHRSignalingChannel.js 库。

在脚本的主体部分中，我们添加了一堆变量，这些变量将用于建立和处理信令通道。虽然它们的用法也许可在此代码后面看出来，但我们在这里要指出两个有趣的变量：haveLocalMedia 和 connected。前者用于跟踪我们是否已成功捕获本地媒体，后者用于跟踪我们是否已通过信令通道（我们的服务器）连接至另一浏览器。只有当两个变量的值都为 true 时，我们才能断定应用程序已设置完毕。通过分别跟踪这两个变量，我们可让两个进程实现并行处理：一方面联系信令服务器，另一方面等待用户提供摄像头和麦克风的使用

许可。稍后我们将讨论执行此操作的代码。

我们还在页面的 onload 处理程序中添加了一些十分有用的代码。如果 URI 以查询参数的形式提供了密钥，例如“http://www.example.org:5001/signaling.html?key=1234”，将在屏幕上的密钥字段中填充此值，然后自动调用 connect() 来连接至服务器，以建立信令通道。

接下来是 connect()，此例程用于建立信令通道。用户可通过单击“Connect”（连接）按钮激活此例程；如果此文件请求的 URI 中以查询参数的形式提供了密钥，则会自动激活此例程（在我们刚刚介绍的 window.onload 中激活）。第一项操作是从屏幕上的密钥输入字段中获取密钥。随后，我们定义将由信令通道使用的处理程序。onWaiting 和 onConnected 处理程序在后面以内嵌方式定义，但 onMessage 处理程序的代码会在后面添加对等媒体协商时大幅增加，因此我们在此处将其单独定义为 handleMsg() 函数。在此阶段，我们的全部工作是将消息追加到远程视频（因为此视频由远程端的浏览器发送，因此称为“远程”视频）下方的消息列表前端。

接下来，我们看看为 onWaiting 和 onConnected 定义的处理程序。建立连接后，我们将状态设置为 connected，并等待本地媒体设置完毕。图 4.14 显示了我们在获取本地媒体之前成功建立连接的视觉效果。请注意，如果我们碰巧是使用给定密钥进行连接的第一浏览器，将调用 onWaiting 处理程序，这时我们将状态设置为 waiting 以便通知用户，并记录我们是等待的一方（如图 4.15 所示）。

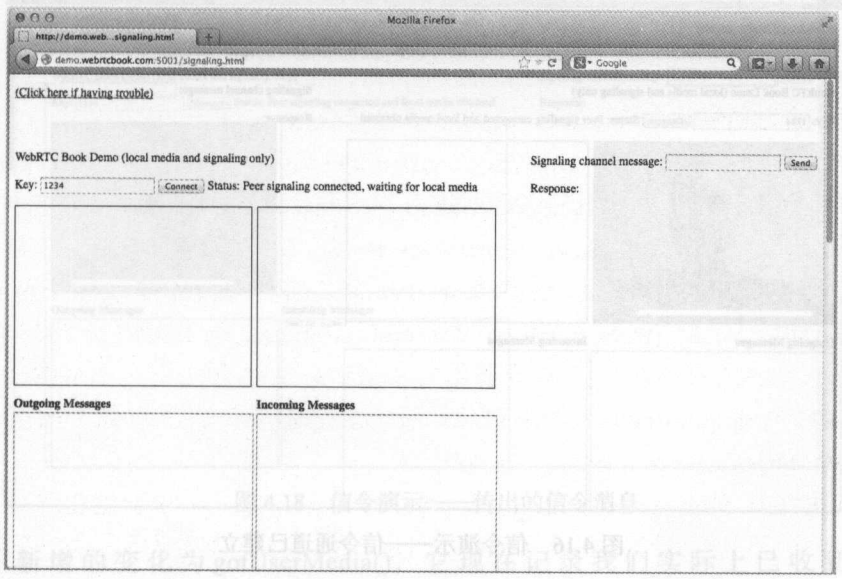


图 4.14 信令演示——正在等待本地媒体

现在，信令通道的处理程序已全部设置完毕，接下来我们可以使用密钥调用 createSignalingChannel() 和处理程序，然后调用其 connect() 方法进行连接。图 4.16 显示了一切（包括本地媒体和信令通道）设置完毕时的显示效果。

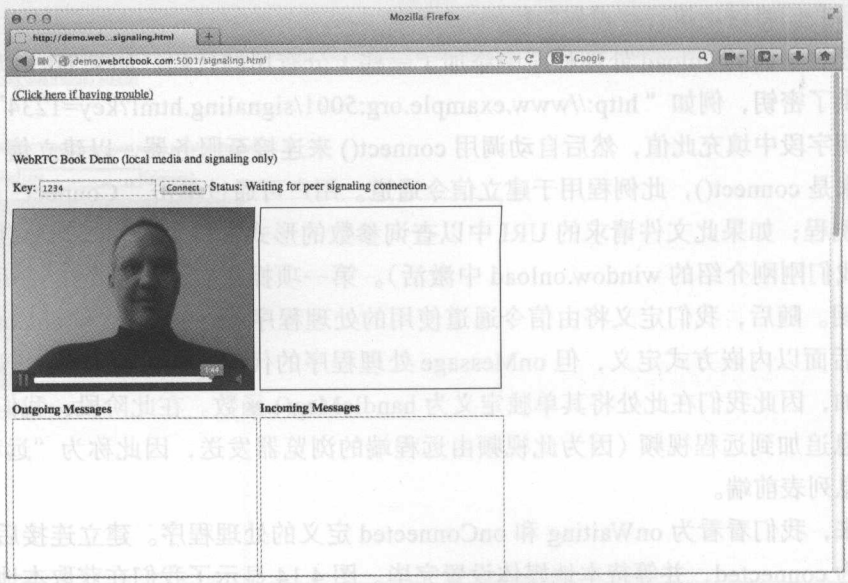


图 4.15 信令演示——正在等待对等端进行连接

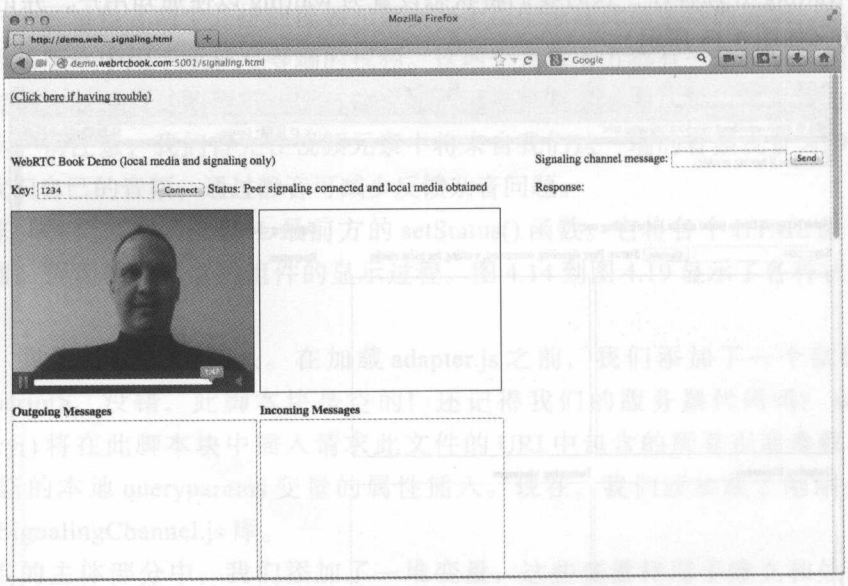


图 4.16 信令演示——信令通道已建立

send() 用于通过信令通道向另一端的浏览器发送消息。如果消息是以参数形式传递的，则发送消息；否则，将从屏幕上的消息字段中提取消息。用户可以通过单击屏幕右上方的“Send”（发送）按钮来显式调用 send()。无论在哪种情况下，都会将该消息追加到浏览器视频下方的消息列表的前端，因为它属于传出的消息。最后一步是通过信令通道发送消息。

图 4.17 显示了一条传入的消息，图 4.18 显示了一条准备通过信令通道发送的消息。

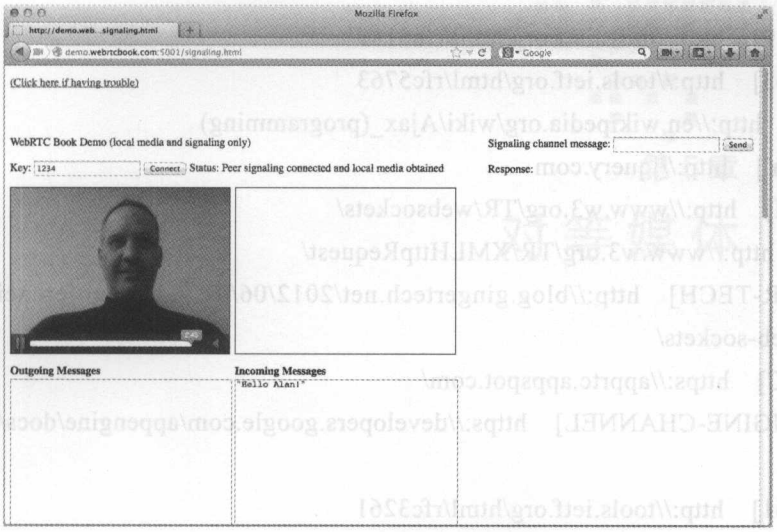


图 4.17 信令演示——传入的信令消息

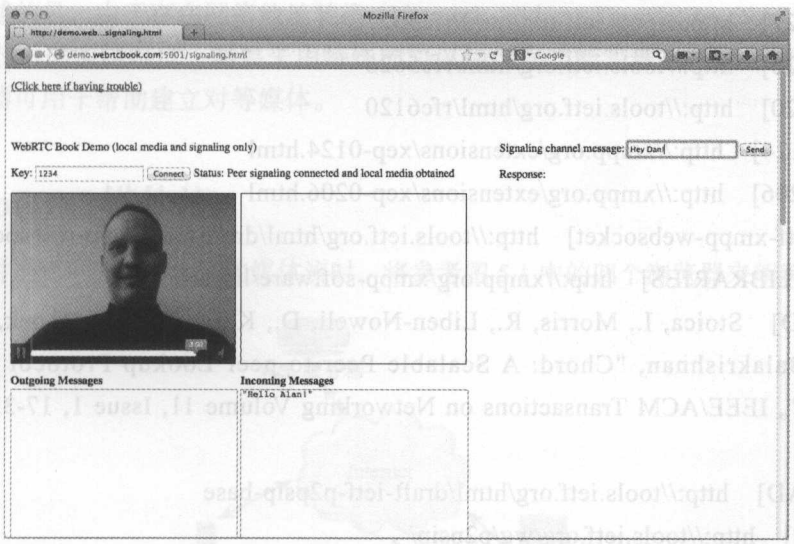


图 4.18 信令演示——传出的信令消息

唯一新增的变化为 gotUserMedia()，它现在记录我们实际上已收到本地媒体（haveLocalMedia = true）这一事件，并通过调用 verifySetupDone() 来结束操作，其中 verifySetupDone() 用于检查信令通道是否已设置完毕。

请注意，在此代码示例中，无需身份验证或授权即可使用服务器的信令通道功能。在编写任何实际应用程序时，你很可能需要设置这两项限制！

4.6 参考资料

[RFC6189] http://tools.ietf.org/html/rfc6189

[RFC5763] http://tools.ietf.org/html/rfc5763

[AJAX] http://en.wikipedia.org/wiki/Ajax_(programming)

[JQUERY] http://jquery.com

[WS-API] http://www.w3.org/TR/websockets/

[XHR] http://www.w3.org/TR/XMLHttpRequest/

[GINGER-TECH] http://blog.gingertech.net/2012/06/04/video-conferencing-in-html5-webrtc-via-web-sockets/

[APPRTC] https://apprtc.appspot.com/

[APP-ENGINE-CHANNEL] https://developers.google.com/appengine/docs/java/channel/overview

[RFC3261] http://tools.ietf.org/html/rfc3261

[RFC7118] http://tools.ietf.org/html/rfc7118

[RFC3327] http://tools.ietf.org/html/rfc3327

[RFC5627] http://tools.ietf.org/html/rfc5627

[RFC5626] http://tools.ietf.org/html/rfc5626

[RFC6120] http://tools.ietf.org/html/rfc6120

[XEP-0124] http://xmpp.org/extensions/xep-0124.html

[XEP-0206] http://xmpp.org/extensions/xep-0206.html

[draft-ietf-xmpp-websocket] http://tools.ietf.org/html/draft-ietf-xmpp-websocket

[XMPP-LIBRARIES] http://xmpp.org/xmpp-software/libraries/

[CHORD] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M., Dabek, F., and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications", IEEE/ACM Transactions on Networking Volume 11, Issue 1, 17-32, Feb 2003, 2001

[RELOAD] http://tools.ietf.org/html/draft-ietf-p2psip-base

[P2PSIP] http://tools.ietf.org/wg/p2psip/

[OPEN-PEER] http://openpeer.org

第 5 章 Chapter 5

对等媒体

WebRTC 采用独特的对等媒体流，其中语音、视频和数据连接都直接在两个浏览器之间建立。遗憾的是，由于存在网络地址转换（Network Address Translation，NAT）和防火墙，增加了这一技术的实施难度，需要采用特殊的协议和过程才能实现。本章介绍的 STUN 和 TURN 服务器可用于帮助建立对等媒体。

5.1 WebRTC 媒体流

本章在讨论两个浏览器之间的媒体流时，将参考图 5.1 中的四个浏览器来阐释相关概念。

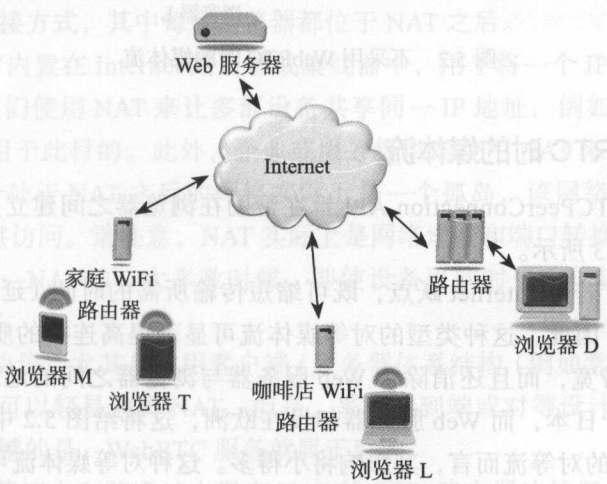


图 5.1 连接到 Internet 的 WebRTC 浏览器

手机和平板电脑通过家中的 WiFi 路由器访问 Internet。笔记本电脑通过咖啡店中的 WiFi 路由器建立连接。PC 则通过公司路由器连接至 Internet。

5.1.1 不采用 WebRTC 时的媒体流

如果不使用 WebRTC 技术或插件，浏览器也能建立媒体流。但是，这些媒体流必须与 Web 浏览通信遵循同一路径。换言之，媒体数据包将先从一端的浏览器流向 Web 服务器，然后再流向另一端的浏览器。图 5.2 显示了这一情况。这样一来，Web 服务器将需要处理额外的流量。高清视频流可占用大量带宽。这就限制了此体系结构的可扩展性。

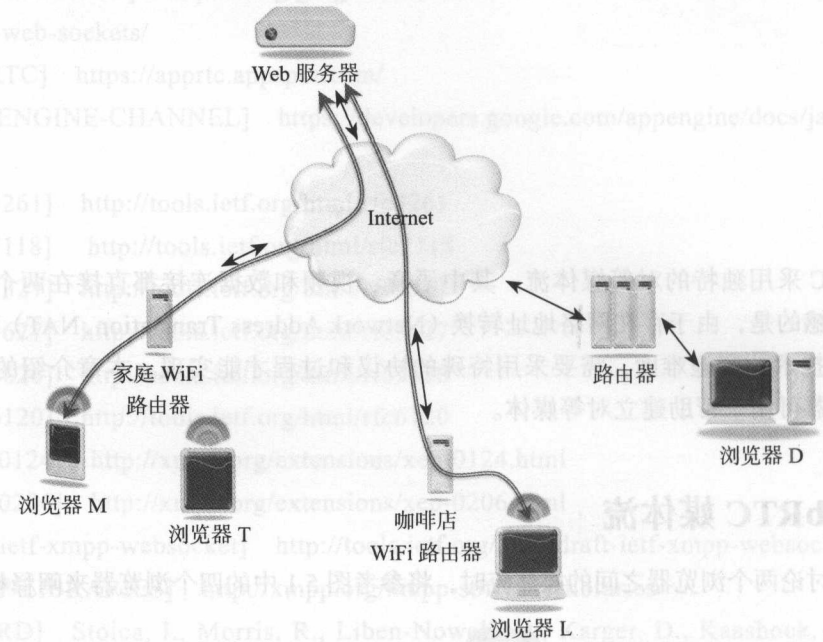


图 5.2 不采用 WebRTC 时的媒体流

5.1.2 采用 WebRTC 时的媒体流

WebRTC 中的 RTCPeerConnection API 旨在帮助在浏览器之间建立直接对等媒体连接。相应的媒体流如图 5.3 所示。

此媒体路径可有若干 Internet 跃点，既可缩短传输所需的时间（延迟较低），又能降低数据包丢失的概率。因此，这种类型的对等媒体流可显著提高连接的质量。它不仅减少了 Web 服务器占用的带宽，而且还消除了 Web 服务器与浏览器之间的地理距离问题。例如，如果两个浏览器位于日本，而 Web 服务器却设在欧洲，这将给图 5.2 中的媒体流带来很大问题，但对图 5.3 中的对等流而言，其影响将小得多。这种对等媒体流可大幅降低提供实时通信服务的成本。

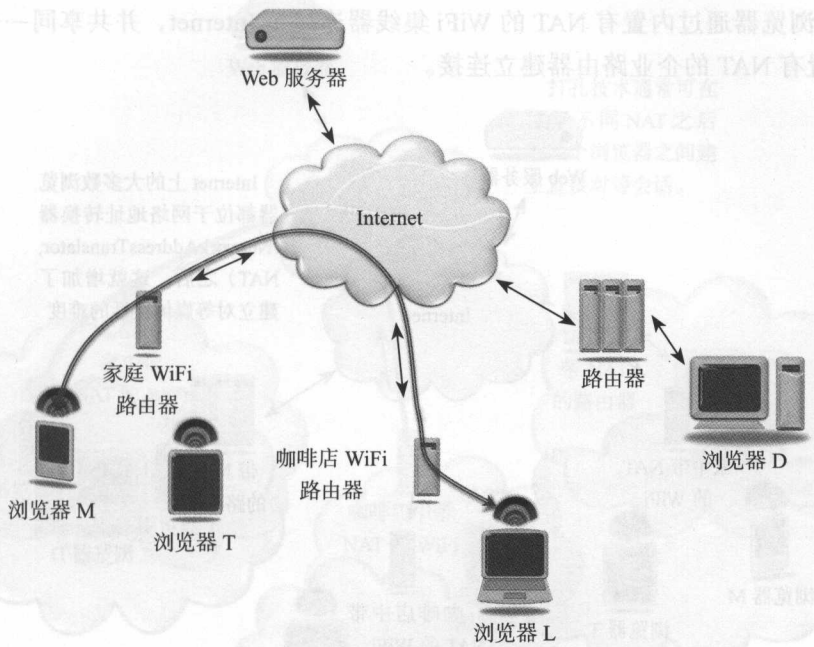


图 5.3 采用 WebRTC 时的对等媒体流

然而，建立这种媒体流实际上非常复杂，因为大多数 Internet 设备都通过 NAT 功能连接至 Internet，详见 5.2 节。

5.2 WebRTC 和网络地址转换

浏览器位于网络地址转换（NAT）设备之后是一种极为普遍的设计。图 5.4 显示了一种更实际的浏览器连接方式，其中每个浏览器都位于 NAT 之后。

NAT 功能通常内置在 Internet 路由器或集线器中，用于将一个 IP 地址空间映射到另一个空间。通常，人们使用 NAT 来让多部设备共享同一 IP 地址，例如家用路由器或集线器中的 NAT 就经常用于此目的。此外，企业或服务提供商使用 NAT 来划分 IP 网段，以简化控制和管理。每个处于 NAT 之后的网络实际上是一个孤岛，该网络中的每个主机完全依赖 NAT 设备来提供访问。请注意，NAT 实际上是网络地址和端口转换器（Network Address and Port Translator，NAPT）。大多数时候，即使设备可同时更改传输地址和传输端口号，人们也称之为 NAT。

许多 Internet 协议，尤其是使用客户端 / 服务器体系结构（例如常规 Web 浏览、电子邮件等）的协议，都可以轻易遍历 NAT。但是，采用端到端或对等设计的协议和服务就会遇到巨大的困难。遗憾的是，WebRTC 服务就属于后者。

在图 5.4 中，笔记本电脑通过内置有 NAT 的 WiFi 路由器连接至 Internet。手机浏览器

和平板电脑浏览器通过内置有 NAT 的 WiFi 集线器连接至 Internet，并共享同一个 IP 地址。PC 使用内置有 NAT 的企业路由器建立连接。

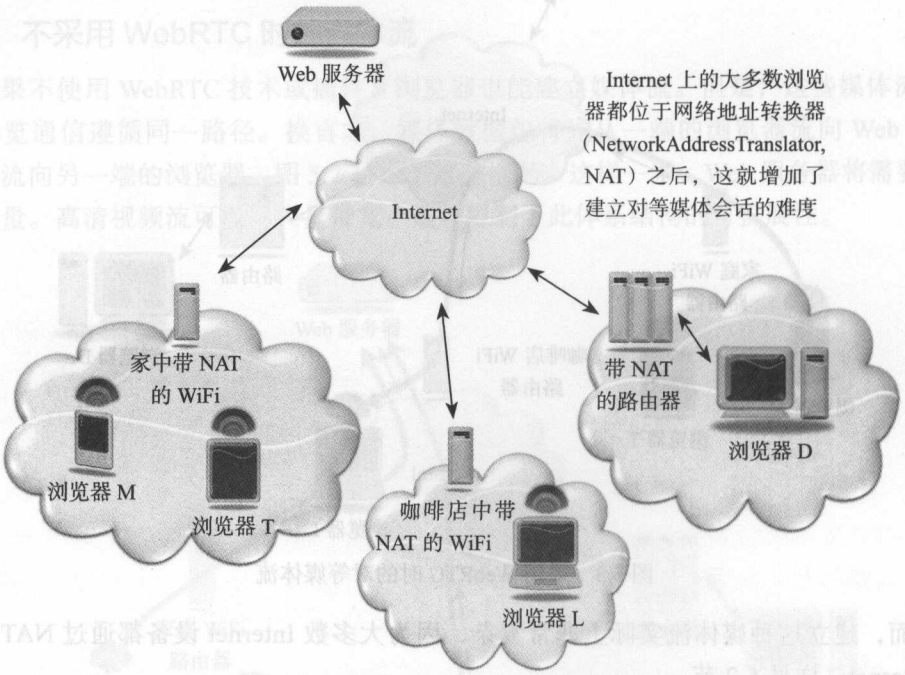


图 5.4　位于 NAT 之后的 WebRTC 浏览器

接下来几节内容将讨论可利用 WebRTC 建立的媒体流类型。其中，有些为跨多个 NAT 的对等媒体流，有些是位于同一 NAT 之后的对等媒体流，还有一些媒体流则由专用 TURN 服务器提供中继。

5.2.1　通过多个 NAT 的对等媒体流

图 5.5 显示了一种可利用 WebRTC 建立的对等媒体流，这里采用了打洞技术（详见第 9 章）。媒体流可绕过 Web 服务器，直接通过多个 NAT 在两个浏览器之间流动。

5.2.2　通过通用 NAT 的对等媒体流

图 5.6 显示了在位于同一 NAT 之后的两个浏览器之间建立媒体会话的情形。在此情形中，最佳媒体路径是保持在局域网之内，而从不越入 Internet。此方式还具有十分出色的质量、带宽和安全属性。与上一种情形一样，需要打洞技术才能实现此媒体流。

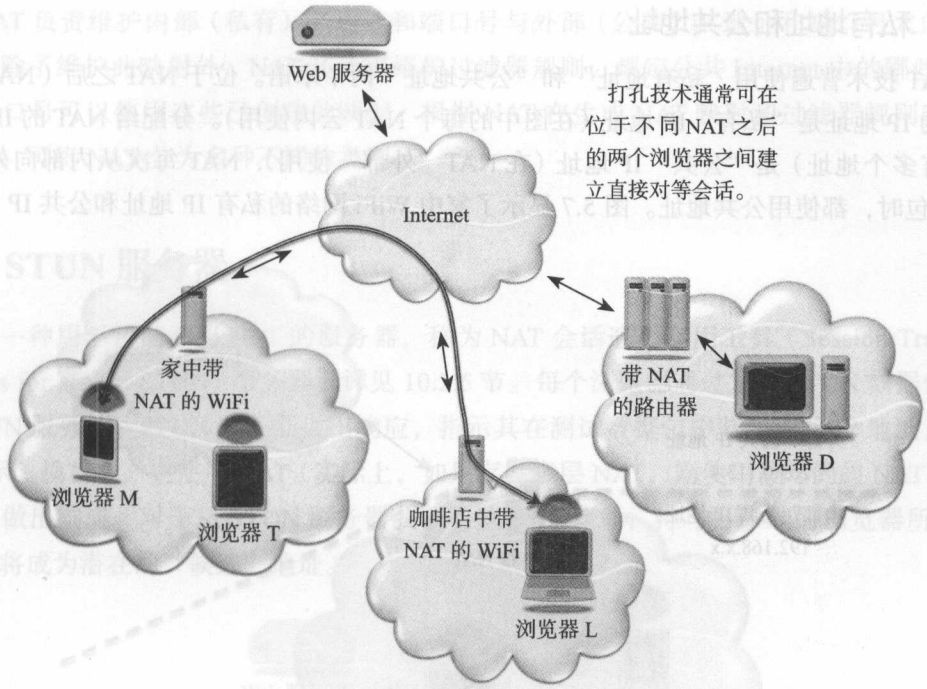


图 5.5 采用 WebRTC 时通过多个 NAT 的对等媒体流

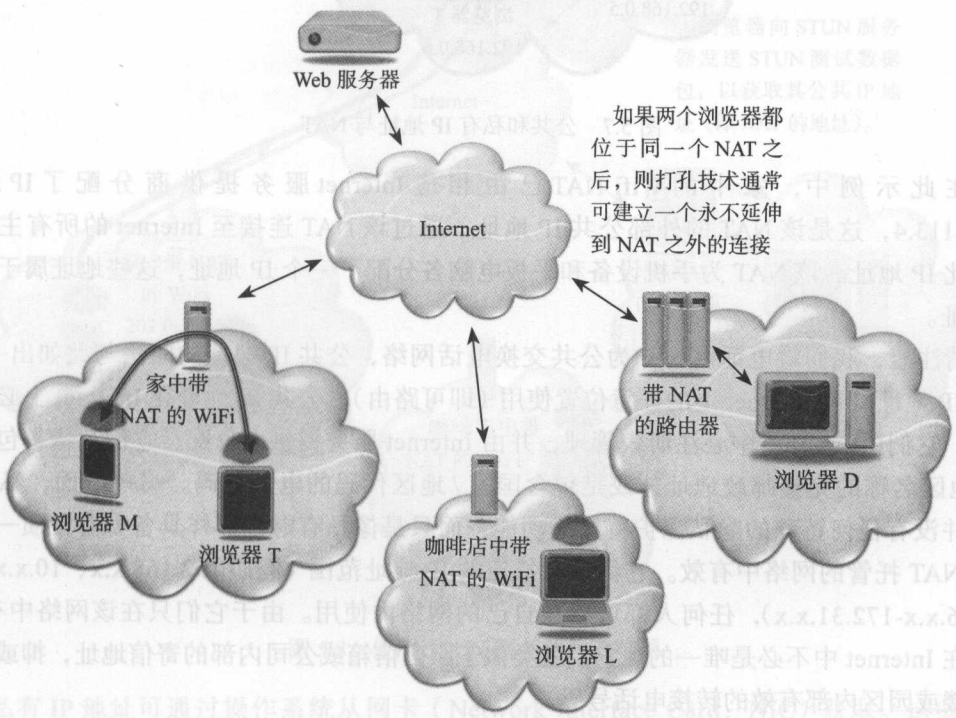


图 5.6 两个浏览器都位于同一 NAT 后面时的媒体流

5.2.3 私有地址和公共地址

NAT 技术普遍使用“私有地址”和“公共地址”两个术语。位于 NAT 之后（NAT“内部”）的 IP 地址是“私有”IP 地址（在图中的每个 NAT 云内使用）。分配给 NAT 的 IP 地址（可能有多个地址）是“公共”IP 地址（在 NAT“外部”使用），NAT 每次从内部向外部转发数据包时，都使用公共地址。图 5.7 显示了家中 WiFi 网络的私有 IP 地址和公共 IP 地址。

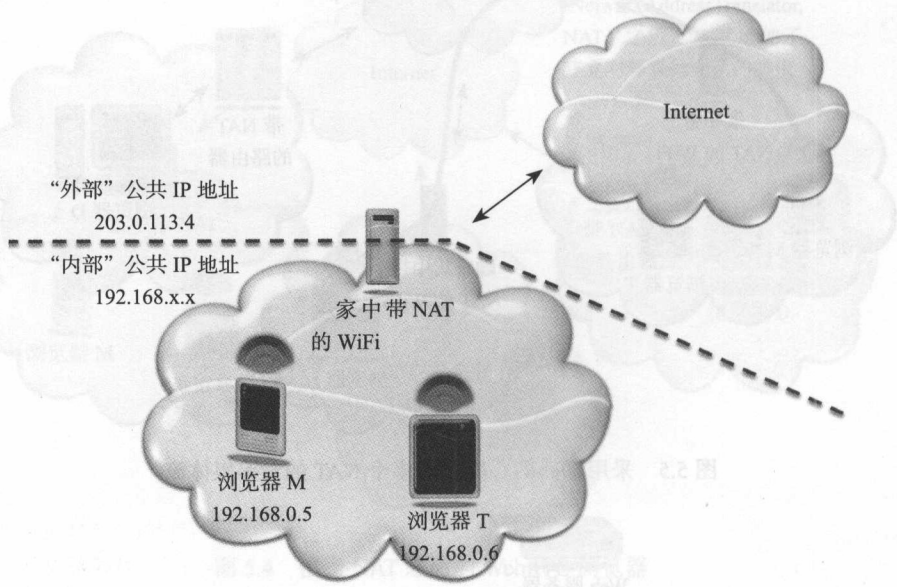


图 5.7 公共和私有 IP 地址与 NAT

在此示例中，家中的 Wifi NAT 已由相应 Internet 服务提供商分配了 IP 地址 203.0.113.4，这是该 NAT 的外部公共 IP 地址，通过该 NAT 连接至 Internet 的所有主机都共享此 IP 地址。该 NAT 为手机设备和平板电脑各分配了一个 IP 地址，这些地址属于私有 IP 地址。

请注意，我们将电话网络称为公共交换电话网络，公共 IP 地址的叫法与此如出一辙。公共 IP 地址可在 Internet 上的任意位置使用（即可路由）。公共 IP 地址在 Internet 上必须是唯一，它们由 Internet 中心注册表管理，并由 Internet 服务提供商分配。它们就像是包含国家 / 地区名称的完整邮政地址，或是包含国家 / 地区代码的电话号码。另一方面，私有 IP 地址并没有任何特殊的隐私保护特性或功能，而只是像私有财产那样具备私有性质——仅在由 NAT 托管的网络中有效。它们拥有特定的 IP 地址范围（例如 192.168.x.x、10.x.x.x 和 172.16.x.x-172.31.x.x），任何人都可以在自己的网络内使用。由于它们只在该网络中有效，因此在 Internet 中不必是唯一的。它们就类似于园区信箱或公司内部的寄信地址，抑或是只在大楼或园区内部有效的转接电话号码。

NAT 负责维护内部（私有）IP 地址和端口号与外部（公共）IP 地址和端口号之间的映射表。除了维护此映射外，NAT 还负责维护过滤器规则，规定公共 Internet 中的哪些 IP 地址和端口号可以使用这些已创建的映射。根据 NAT 在生成 NAT 映射和过滤器规则时应用的规则，可将 NAT 分为多种不同的类别。

5.3 STUN 服务器

有一种用于帮助遍历 NAT 的服务器，称为 NAT 会话遍历实用工具（Session Traversal Utilities for NAT，STUN）服务器，详见 10.2.5 节。每个浏览器通过发送 STUN 数据包来查询 STUN 服务器。STUN 服务器做出响应，指示其在测试数据包中监测到的 IP 地址，如图 5.8 所示。换言之，它使用 NAT（实际上，如果存在多层 NAT，则使用最外侧的 NAT）映射的地址做出响应。对于从 STUN 服务器获取的这一 IP 地址，将与另一端的浏览器所共享，此地址将成为潜在的“候选”地址。

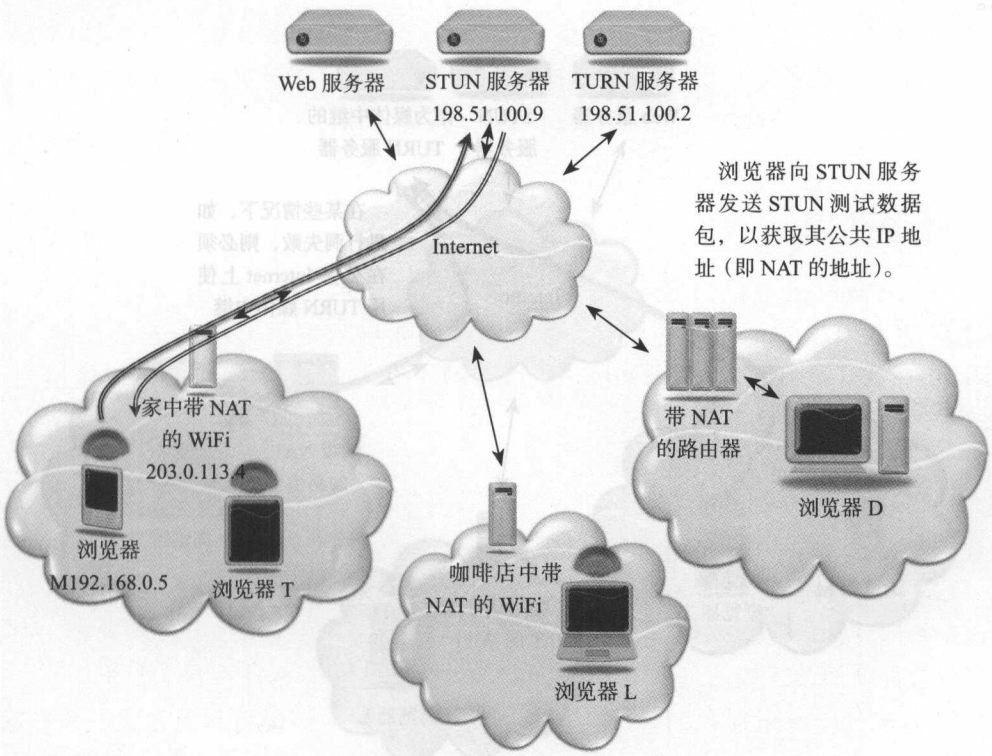


图 5.8 使用 STUN 服务器的浏览器

私有 IP 地址可通过操作系统从网卡（Network Interface Card，NIC）获取。这些地址

可以是 IPv4 或 IPv6，抑或是二者的组合。此外，虚拟专用网络（Virtual Private Network，VPN）连接还可能提供额外的地址，也可以使用其他 NAT 遍历协议，例如通用即插即用（Universal Plug and Play，UPnP），不过这类协议并不常见。

5.4 TURN 服务器

还有一种用于帮助遍历 NAT 的服务器，名为使用中继型 NAT 遍历（Traversal Using Relay around NAT，TURN）服务器，详见 10.2.6 节。浏览器通过查询 TURN 服务器来获取媒体中继地址。媒体中继地址是一个公共 IP 地址，用于转发从浏览器收到的数据包，或者将收到的数据包转发给浏览器。如果两个对等端之间单纯因为 NAT 的类型而无法建立直接的对等媒体会话，则可以使用中继地址。

虽然此媒体流并不理想，但至少不像图 5.2 中所示的那样通过 Web 服务器提供媒体中继。此外，这只出现在所有直接媒体路径都失败且没有替代方案的少数情况下，如图 5.9 所示。

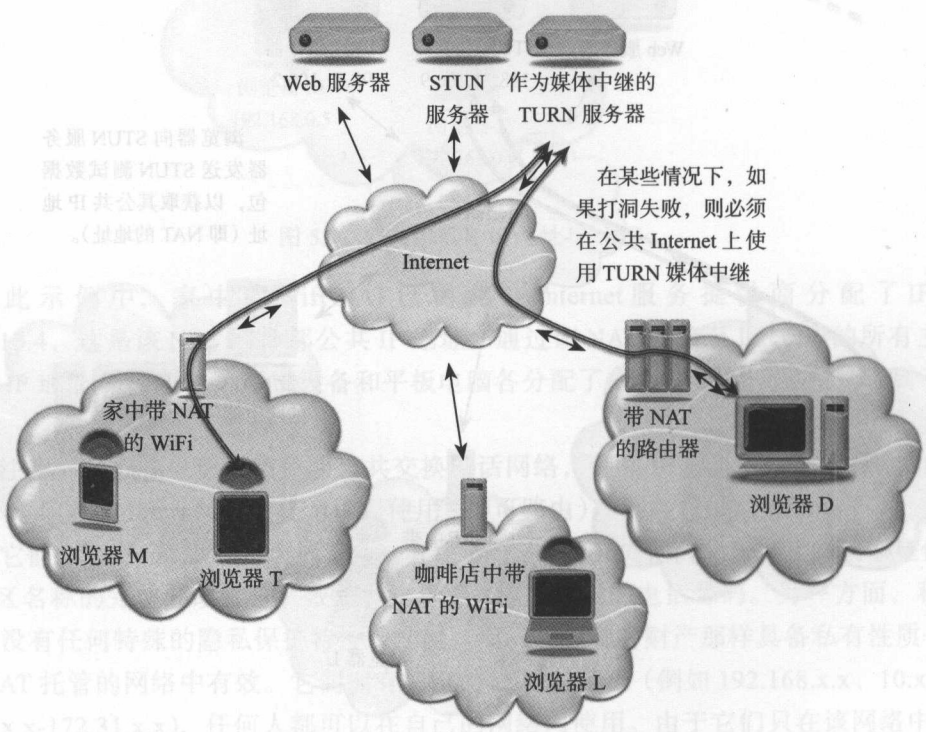


图 5.9 由 TURN 服务器提供中继的媒体

5.5 候选项

打洞技术（详见第9章）依靠将要建立的会话中的每个对等端来收集一组可通过Internet访问它们的潜在方式。这些IP地址和端口组称为“地址候选项”，简称“候选项”。借助STUN服务器，浏览器可识别自己是否位于NAT之后以及该NAT的IP地址（此地址称为“反射候选项”）。利用TURN服务器，浏览器可获取中继地址，即公共Internet上用于向浏览器转发通信的IP地址和端口。此地址称为“中继候选项”。在JavaScript中，这些候选项包含在RTCIceCandidates对象的candidate属性中。

用于实现打洞的协议称为交互式连接建立（Interactive Connectivity Establishment，ICE）协议（详见第9章）。

Chapter 6 第 6 章

对等连接和提议 / 应答协商

WebRTC 标准定义了两组主要的功能：一是媒体捕获（详见第 3 章）；二是媒体传输（详见本章）。对等连接和提议 / 应答协商的概念是建立 WebRTC 对等媒体和数据的核心。本章将详细介绍这两个关键要素——对等连接和提议 / 应答协商。

6.1 对等连接

RTCPeerConnection 接口是 WebRTC 技术的主要 API。此 API 的功能是在两个浏览器之间建立媒体和数据连接路径。虽然此 API 紧密绑定到 JavaScript 会话建立协议（JavaScript Session Establishment Protocol，JSEP）（详见 11.3.8 节）以进行媒体协商，但 JSEP 的大多数细节都由浏览器处理。

WebRTC 规范中给出的对等连接定义有点让人困惑。RTCPeerConnection 接口定义了如何创建对等连接，稍后我们就会发现，这一定义十分琐细。不过，RTCPeerConnection 接口中还定义了其他几个 API：数据通道创建 API、DTMF 启用和控制 API、连接统计数据 API，以及（用户）对等端身份确定和验证控制 API。本章只讨论对等连接和媒体协商。

说来奇怪，WebRTC 对等连接并不是连接，至少不是 TCP 意义上的那种连接。它是一组路径建立进程（ICE）以及一个可确定应建立哪些媒体和数据路径的协商器。

RTCPeerConnection 对象的构造函数有一个配置对象，该配置对象包含一系列属性，其中最重要的是 iceServers，此属性是一个服务器地址列表，用于帮助通过 NAT 和防火墙（STUN 和 TURN 服务器，详见 10.2.5 节和 10.2.6 节）建立会话：

```
// 创建对等连接
```

```
pc = new RTCPeerConnection(
        {iceServers:[{"url":"stun:stun.l.google.com:19302"},
                     {"url":"turn:user@turn.myserver.com",
                      "credential":"test"}]});
```

要让你的本地 MediaStream 传输至另一端的浏览器，下一步就是向你的浏览器做出这一指示。用于执行此操作的方法是 addStream()：

```
// 获取本地音频和视频
getUserMedia({"audio":true, "video":true}, successCB, failureCB);

function successCB(myStream) {
  // 告诉我的浏览器，我要发送 MediaStream
  pc.addStream(myStream);
}
```

可以看到，在对等连接中“添加”MediaStream 非常简单。令人吃惊的是，此操作不会产生任何媒体流。实际上，addStream() 的作用只是告诉浏览器，你希望浏览器与对等端就 MediaStream 发送问题进行协商。与 addStream() 相对应的是 removeStream() 方法，但是人们正在讨论是否有可能去掉这个方法，因为用户很少在不想移除整个对等连接的情况下移除流。

那么，我们如何让媒体流动起来呢？在介绍实现媒体流动的方法之前，我们需要先来回顾一下提议 / 应答协商。

6.2　提议 / 应答协商

要在双方之间建立媒体会话，必须在二者之间协商会话。协商之所以不可或缺，是因为双方需要为会话确定一组通用的功能特性。这组功能特性事先并不明确，因为它取决于双方使用的具体浏览器版本、浏览器上运行的 JavaScript 以及用户双方各自的选择。WebRTC 中使用的协商方式称为“提议 / 应答”，这是一种“一次性通过”型协商机制。首先，一方针对要建立的媒体会话类型创建描述，以此来发起媒体会话，此过程称为“提议”。第一方使用信令通道将提议发送给另一方。随后，另一方予以回应，此过程称为“应答”。此应答将列出，在第一方提议的会话功能特性中，另一方能够或愿意在此会话中支持或使用哪些功能特性。应答将被发送回第一方，并且也通过信令通道传输。从生成提议到收到应答的整个交换过程就称为“提议 / 应答交换”或“提议 / 应答协商”。

提议 / 应答交换可确保双方都知道要发送和接收的媒体类型，以及如何正确解码和处理该媒体。这包括如下信息：要使用的一个或多个编解码器、这些编解码器的参数、用于媒体加密和身份验证的密钥信息，等等。此外，通过提议 / 应答交换，双方还可以就以下两点达成一致：一是要建立的媒体类型（音频、视频或数据通道），二是每种媒体有多少流量将通过相同的传输地址进行多路传输。此外，还会交换有关媒体流的标识信息，例如媒

体流 ID（Media Stream ID，MSID），此 ID 用于将 JavaScript 操作绑定到收到的媒体。请注意，在实际中，可能需要进行多次提议 / 应答交换才能协商好一切。例如，如果要对多个 MediaStreamTrack 进行多路传输并通过单个端口统一发送，则可能需要多次提议 / 应答交换才能安排妥当。

WebRTC 使用 RTCSessionDescription 对象来表示提议和应答，该对象是会话描述的容器。每个浏览器都将生成一个 RTCSessionDescription 对象，并通过信令通道从另一端的浏览器接收一个 RTCSessionDescription 对象。有关 RTCSessionDescription 中如何使用会话描述协议的详细信息，请参见 12.2.1 节。下一节将讨论 JavaScript 如何生成和使用 RTCSessionDescription 对象。

6.3 JavaScript 提议 / 应答控制

安排此提议 / 应答协商涉及一系列编程步骤。实际上，本地浏览器只关注两个特定的调用：

```
// 将我的会话描述告诉我的浏览器
pc.setLocalDescription(mySessionDescription);

// 将对等端的会话描述告诉我的浏览器
pc.setRemoteDescription(yourSessionDescription);
```

在此代码中，mySessionDescription 对象从我的浏览器角度描述了媒体流。换言之，此特定的会话描述可能不仅描述了我希望发送的内容，还描述了我希望接收的内容。与此相反，yourSessionDescription 对象从另一端浏览器的角度描述了媒体流。此外，它不仅描述了该对等端希望发送的内容，也描述了它希望接收的内容。如果二者根据 SDP 协商规则彼此兼容（请阅读 12.2.2.1 节和 12.2.2.2 节，查看实际浏览器 SDP 提议和应答的示例），则协商成功，媒体随即可以开始流动。请注意，其中一个将是提议，另一个则是应答，但任何一个都可以作为我们的本地描述，另一个则作为远程描述。对于只有一个提议和一个应答的协商，其能否成功取决于二者是否兼容；如果兼容，其中之一将设置为本地描述，另一个则设置为远程描述。由于这些提议和应答 RTCSessionDescription 对象的语法十分复杂，而 WebRTC 的原则是向 Web 开发人员隐藏尽可能多的复杂性，因此 WebRTC 为开发人员提供了特殊的方法，用于使浏览器自动生成提议和应答：

```
// 生成提议
pc.createOffer(gotOffer, didntGetOffer);
function gotOffer(aSessionDescription) {
  // 这很简单。
  setLocalDescription(aSessionDescription);
  // 现在将会话描述（提议）发送给对等端，以便对等端：
```

```
  // a) 将提议传递给 setRemoteDescription 并且 b) 调用 createAnswer，如下所示
}

//////// 或者 ////////

// 生成应答（需要已经对提议调用 setRemoteDescription）
pc.createAnswer(gotAnswer, didntGetAnswer);
function gotAnswer(aSessionDescription) {
  // 这着实简单。
  setLocalDescription(aSessionDescription);
  // 现在将会话描述（应答）发送给对等端，以便对等端
  // 将应答传递给 setRemoteDescription。
}

// 请注意，对于每次特定的协商，都将只生成 * 一个 * 提议 / 应答！
// 任何生成提议的对等端将启动协商。
// 任何从该对等端收到提议的另一端必须生成应答并发送回前一对等端
```

上面这段代码看起来很复杂，其原因有二：

1. 它同时显示了呼叫和应答代码。

2. 它显示了需要并行执行的两个步骤：a）创建提议或应答；b）设置本地和远程描述。

当你在下一节的示例上下文中查看这段代码时，就比较容易理解了。但在查看该代码之前，我们需要先说明其他几个问题。关于何时生成应答，很容易就可以判断出来——在从对等端收到提议时生成。但是，何时生成提议呢？从根本上讲，只有浏览器知道何时需要进行新的提议 / 应答协商，WebRTC 提供了 negotiationneeded 事件和关联的 onnegotiationneeded 处理程序，通过定义它们可生成提议等对象。每当浏览器识别到需要进行媒体协商的变化时，就会执行该处理程序。这些变化包括：你的应用程序调用了 addStream()；远程对等端对流进行了更改；发生了某种媒体故障，而浏览器识别到可通过新的协商来加以解决。简而言之，你的代码应当设置 onnegotiationneeded 处理程序，才能为新的媒体协商提供可靠的调用。为便于说明，本书中的示例代码从不这样设置，而是采取以下处理方式：只要通过 addStream() 添加媒体并且存在可供交换提议和应答的信令通道，就从一端生成提议。

这里需要注意的另一个重要问题是，不存在标准化方法来交换提议和应答。此工作由你的代码负责，你的信令通道代码必须非常精确。如果你不清楚这是什么意思，则一定是跳过了第 4 章，建议你回过头去阅读该章的内容！

最后，将有大量处理程序和状态属性可用于跟踪提议 / 应答交换的状态，你可能希望使用它们来改善应用程序的用户友好性，以便在简单的提议 / 应答协商因某种原因而失败时采取相应的处理。第 8 章简要列出了其中的许多处理程序和属性，但请注意，许多处理程序和属性的具体名称，尤其是可以使用的值，目前仍未最终确定。

6.4 可运行的代码示例：对等连接和提议 / 应答协商

下面，我们将在第 4 章给出的示例的基础上，使用 WebRTC 的提议 / 应答功能加入用于对等连接和媒体协商的代码。两个浏览器通过二者之间的信令通道发送 SDP 提议和应答，以此来协商媒体连接。随后，两个浏览器将尽可能直接在双方之间发送媒体。

这次唯一的差异位于 HTML 文件中。下一节将对此进行介绍。

客户端 WebRTC 应用程序

```
<!--
// 版权所有 2013-2014 Digital Codex LLC
// 你可以将此代码用于自身学习
// 如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，
// 请勿标榜你的代码先于此代码问世
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责
-->

<html>
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
  <style>
    video {
      width:  320px;
      height:  240px;
      border:  1px solid black;
    }
    div {
      display:  inline-block;
    }
  </style>
</head>
<body>

<!-- 此空白脚本部分专为查询参数预留 -->
<script></script>

<!-- 加载 polyfill，先加载本地副本以进行本地测试 -->
<script src="extra/adapter.js" type="text/javascript"></script>
<script
  src="https://webrtc.github.io/adapter/adapter.js"
  type="text/javascript"></script>

<!-- 加载基于 XHR 的信令通道，以基于密钥定向连接 -->
<script src="clientXHRSignalingChannel.js"
        type="text/javascript"></script>
```

```
<script>
var signalingChannel, key, id,
    haveLocalMedia = false,
    weWaited = false,
    myVideoStream, myVideo,
    yourVideoStream, yourVideo,
    doNothing = function() {},
    pc,
    constraints = {mandatory: {
                      OfferToReceiveAudio: true,
                      OfferToReceiveVideo: true}};

////////////////////////////
// 这是主例程
////////////////////////////

// 由此开始获取本地媒体。
// 此外，它还可以自动启动信令通道
window.onload = function () {

  // 如果URI中提供了密钥，则自动连接信令通道
  if (queryparams && queryparams['key']) {
    document.getElementById("key").value = queryparams['key'];
    connect();
  }

  myVideo = document.getElementById("myVideo");
  yourVideo = document.getElementById("yourVideo");

  getMedia();

  // 如果建立连接，connect()将调用createPC()。
  // 如果createPC()和getMedia()成功执行，将调用attachMedia()
};

////////////////////
// 接下来这一节代码用于建立信令通道
////////////////////

// 此例程会连接至Web服务器并建立信令通道
// 当加载文档或用户单击"Connect"（连接）按钮时，将自动调用此例程
function connect() {
  var errorCB, scHandlers, handleMsg;

  // 首先，获取用于连接的密钥
  key = document.getElementById("key").value;

  // 此处理程序用于处理通过信令通道收到的所有消息。
  handleMsg = function (msg) {
```

```
    // 首先，我们整理消息并将其发布到屏幕上
    var msgE = document.getElementById("inmessages");
    var msgString = JSON.stringify(msg).replace(/\\r\\n/g,'\n');
    msgE.value = msgString + "\n" + msgE.value;

    // 随后，我们基于消息的类型执行操作
    if (msg.type === "offer") {
      pc.setRemoteDescription(new RTCSessionDescription(msg));
      answer();
    } else if (msg.type === "answer") {
      pc.setRemoteDescription(new RTCSessionDescription(msg));
    } else if (msg.type === "candidate") {
      pc.addIceCandidate(
        new RTCIceCandidate({sdpMLineIndex:msg.mlineindex,
                             candidate:msg.candidate}));
    }
  };

  // 用于信令通道的处理程序
  scHandlers = {
    'onWaiting' : function () {
      setStatus("Waiting");
      // 后面将使用 weWaited 执行自动调用
      weWaited = true;
    },
    'onConnected': function () {
      setStatus("Connected");
      // 由于我们已成功连接，因此建立 RTC 对等连接
      createPC();
    },
    'onMessage': handleMsg
  };

  // 最后，创建信令通道
  signalingChannel = createSignalingChannel(key, scHandlers);
  errorCB = function (msg) {
    document.getElementById("response").innerHTML = msg;
  };

  // 并进行连接
  signalingChannel.connect(errorCB);
}


// 此例程通过信令通道发送消息
// 其方式有二：一是执行显式调用；二是通过用户单击 "Send"（发送）按钮
function send(msg) {
  var handler = function (res) {
    document.getElementById("response").innerHTML = res;
    return;
```

```
  },

  // 如果没有传入，则获取消息
  msg = msg || document.getElementById("message").value;

  // 整理消息并发布到屏幕上
  msgE = document.getElementById("outmessages");
  var msgString = JSON.stringify(msg).replace(/\\r\\n/g,'\n');
  msgE.value = msgString + "\n" + msgE.value;

  // 同时通过信令通道发送
  signalingChannel.send(msg, handler);
}

////////////////////////////
// 接下来这一节代码用于获取本地媒体
////////////////////////////

function getMedia() {
  getUserMedia({"audio":true, "video":true},
               gotUserMedia, didntGetUserMedia);
}

function gotUserMedia(stream) {
  myVideoStream = stream;
  haveLocalMedia = true;

  // 向我显示我的本地视频
  attachMediaStream(myVideo, myVideoStream);
  // 等待 RTCPeerConnection 创建完毕
  attachMediaIfReady();
}

function didntGetUserMedia() {
  console.log("couldn't get video");
}

////////////////////////////
// 接下来这一节代码用于建立 RTC 对等连接
////////////////////////////

function createPC() {
  var stunuri = true,
      turnuri = false,
      myfalse = function(v) {
                  return ((v==="0")||(v==="false")||(!v)); },
      config = new Array();

  // 基于各个查询参数调整配置字符串
```

```
  if (queryparams) {
    if ('stunuri' in queryparams) {
      stunuri = !myfalse(queryparams['stunuri']);
    }
    if ('turnuri' in queryparams) {
      turnuri = !myfalse(queryparams['turnuri']);
    };
  };

if (stunuri) {
    // 这是Google的一台公共STUN服务器
    config.push({"url":"stun:stun.l.google.com:19302"});
  }
  if (turnuri) {
    if (stunuri) {
      // 这里不能使用仅支持TURN的TURN服务器，因为Chrome中存在一个Bug，
      // 可导致STUN服务器的响应遭到忽略，
      // 因此我们使用同时具备STUN功能的TURN服务器
      config.push({"url":"turn:user@turn.webrtcbook.com",
                   "credential":"test"});
    } else {
      // 这是我们仅支持TURN的TURN服务器
      config.push({"url":"turn:user@turn-only.webrtcbook.com",
                   "credential":"test"});
    }
  }
  console.log("config = " + JSON.stringify(config));

  pc = new RTCPeerConnection({iceServers:config});
  pc.onicecandidate = onIceCandidate;
  pc.onaddstream = onRemoteStreamAdded;
  pc.onremovestream = onRemoteStreamRemoved;

  // 等待本地媒体准备就绪
  attachMediaIfReady();
}

// 如果我们的浏览器有另一个候选项，则将其发送给对等端
function onIceCandidate(e) {
  if (e.candidate) {
    send({type: 'candidate',
          mlineindex: e.candidate.sdpMLineIndex,
          candidate: e.candidate.candidate});
  }
}

// 如果我们的浏览器检测到另一端添加了媒体流，则将其显示在屏幕上
function onRemoteStreamAdded(e) {
  yourVideoStream = e.stream;
  attachMediaStream(yourVideo, yourVideoStream);
```

```
  setStatus("On call");
}

// 没错，如果远程端移除该流，我们不执行任何操作。
// 毕竟这是一个 * 简单 * 的演示。
function onRemoteStreamRemoved(e) {}


////////////////////////////////////
// 接下来这一节代码用于将本地媒体附加到对等连接
////////////////////////////////////

// 此守护例程实际上用于对两项异步活动的完成时间进行同步：
// 一是创建对等连接；二是获取本地媒体
function attachMediaIfReady() {
  // 如果 RTCPeerConnection 已经就绪，
  // 并且我们已获得本地媒体，则继续处理。
  if (pc && haveLocalMedia) {attachMedia();}
}

// 此例程将我们的本地媒体流添加至对等连接
// 请注意，这不会导致任何媒体开始流动
// 其作用只是指示浏览器在其下一个 SDP 描述中加入此流
function attachMedia() {
  pc.addStream(myVideoStream);
  setStatus("Ready for call");

  // 如果 URI 中 call 参数的值表示 true，则自动执行调用，
  // 但还要确保我们已完成连接之前的所有步骤（提高两端已一切就绪的概率）
  if (queryparams && queryparams['call'] && !weWaited) {
    call();
  }

}


////////////////////////////
// 接下来这一节代码用于呼叫和应答
////////////////////////////

// 以下代码为提议生成会话描述
function call() {
  pc.createOffer(gotDescription, doNothing, constraints);
}

// 以下代码为应答生成会话描述
function answer() {
  pc.createAnswer(gotDescription, doNothing, constraints);
}

// 无论在哪一种情况下，一旦我们获取会话描述，
```

```
// 就指示我们的浏览器将其用作本地描述，
// 然后将其发送给另一端的浏览器
// 只有先设置了本地描述，浏览器才能发送媒体并准备从另一端接收媒体
function gotDescription(localDesc) {
  pc.setLocalDescription(localDesc);
  send(localDesc);
}


////////////////////////////////////
// 这一节代码用来基于应用程序的进度更改 UI。
////////////////////////////////////

// 此函数将通过隐藏、显示和填充各种 UI 元素，
// 让用户大体了解浏览器在建立信令通道、
// 获取本地媒体、创建对等连接
// 以及实际连接媒体（呼叫）方面的进度。
function setStatus(str) {
  var statuslineE = document.getElementById("statusline"),
      statusE = document.getElementById("status"),
      sendE = document.getElementById("send"),
      connectE = document.getElementById("connect"),
      callE = document.getElementById("call"),
      scMessageE = document.getElementById("scMessage");

  switch (str) {
    case 'Waiting':
      statuslineE.style.display = "inline";
      statusE.innerHTML =
        "Waiting for peer signaling connection";
      sendE.style.display = "none";
      connectE.style.display = "none";
      break;
    case 'Connected':
      statuslineE.style.display = "inline";
      statusE.innerHTML =
        "Peer signaling connected, waiting for local media";
      sendE.style.display = "inline";
      connectE.style.display = "none";
      scMessageE.style.display = "inline-block";
      break;
    case 'Ready for call':
      statusE.innerHTML = "Ready for call";
      callE.style.display = "inline";
      break;
    case 'On call':
      statusE.innerHTML = "On call";
      callE.style.display = "none";
      break;
    default:
```

```
    }
  }
</script>

<div id="setup">
  <p>WebRTC Book Demo (local media, signaling, and peer connection only)</p>
  <p>Key:
    <input type="text" name="key" id="key"
           onkeyup="if (event.keyCode == 13) {
                    connect(); return false;}"/>
    <button id="connect" onclick="connect()">Connect</button>
    <span id="statusline" style="display:none">Status:
      <span id="status">Disconnected</span>
    </span>
    <button id=" call" style=" display:none"
            onclick = "call()" >Call</button>
  </p>
</div>

<div id="scMessage" style="float:right;display:none">
  <p>Signaling channel message:
    <input type="text" width="100%" name="message" id="message"
           onkeyup="if (event.keyCode == 13) {
                    send(); return false;}"/>
    <button id="send" style="display:none"
            onclick="send()">Send</button>
  </p>

  <p>Response:  <span id="response"></span></p>
</div>

<br/>

<div style="width:30%;vertical-align:top">
  <div>
    <video id="myVideo" autoplay="autoplay" controls
           muted="true"/>
  </div>
  <p><b>Outgoing Messages</b>
    <br/>
    <textarea id="outmessages" rows="100"
              style="width:100%"></textarea>
  </p>
</div>

<div style="width:30%;vertical-align:top">
  <div>
    <video id="yourVideo" autoplay="autoplay" controls />
  </div>
```

```
    <p><b>Incoming Messages</b>
      <br/>
      <textarea id="inmessages" rows="100"
                style="width:100%"></textarea>
    </p>
  </div>
  </body>
  </html>
```

我们还是先来看末尾的 HTML 标记。此版本唯一增加的内容是“ Call”（呼叫）按钮。在做好一切准备来执行呼叫之前，此按钮一直处于隐藏状态。图 6.1 提供了此版本在连接之前的显示效果，这十分类似于我们之前的迭代版本。但在建立信令连接之后，其外观就发生了变化，大家可以在图 6.2 中看到，它现在多了一个新的“ Call”（呼叫）按钮。现在请注意，SDP 消息将显示在视频元素下方的信令消息窗口中，且最新的消息显示在最上方。在介绍信令代码时，我们曾经指出，来自我们这一端的音频在视频元素中设置为静音。尽管我们自己的音频已经静音，但仍建议使用耳机，以免来自对等端的音频反过来进入设备上的麦克风，这种情况很容易出现在笔记本电脑上。

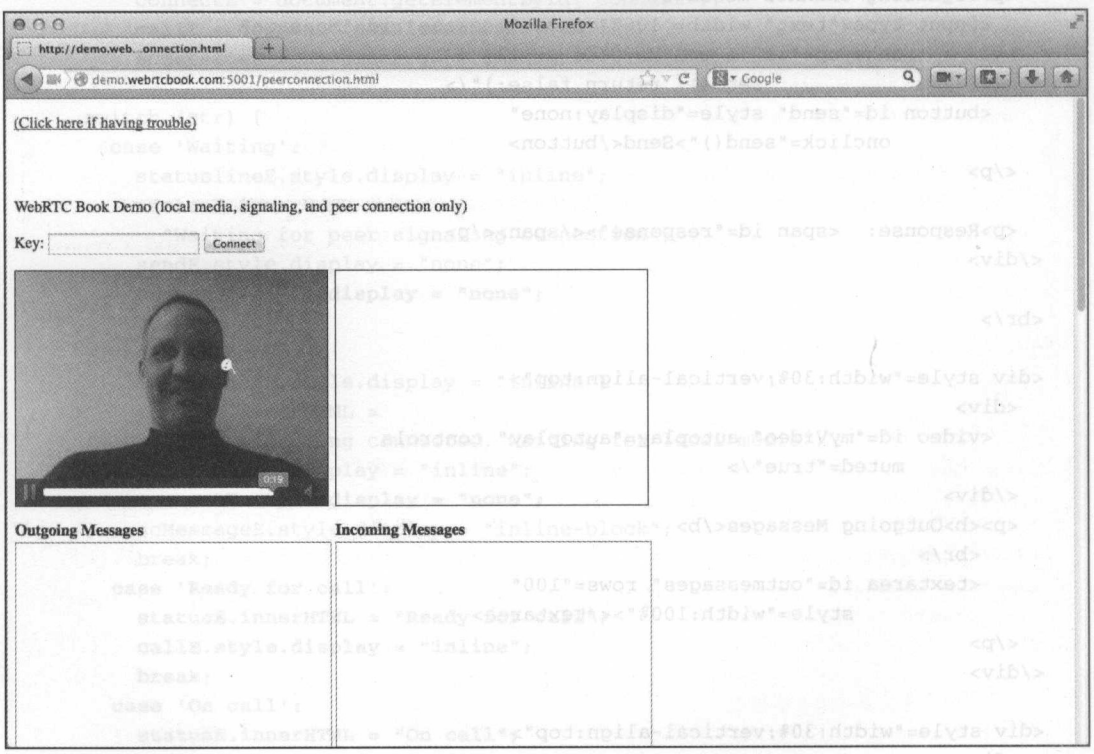


图 6.1　对等连接演示

在转到文件开头之前，我们先来快速看一下 HTML 最前方的 setStatus() 函数有哪些变

化。我们的状态已经从“Set up”(准备) 变为“Ready for call”(呼叫就绪)，这会启用“Call”(呼叫) 按钮。此外，当创建对等连接并完成媒体提议 / 应答协商之后，还会激活一种新的状态“On call”(通话中)。

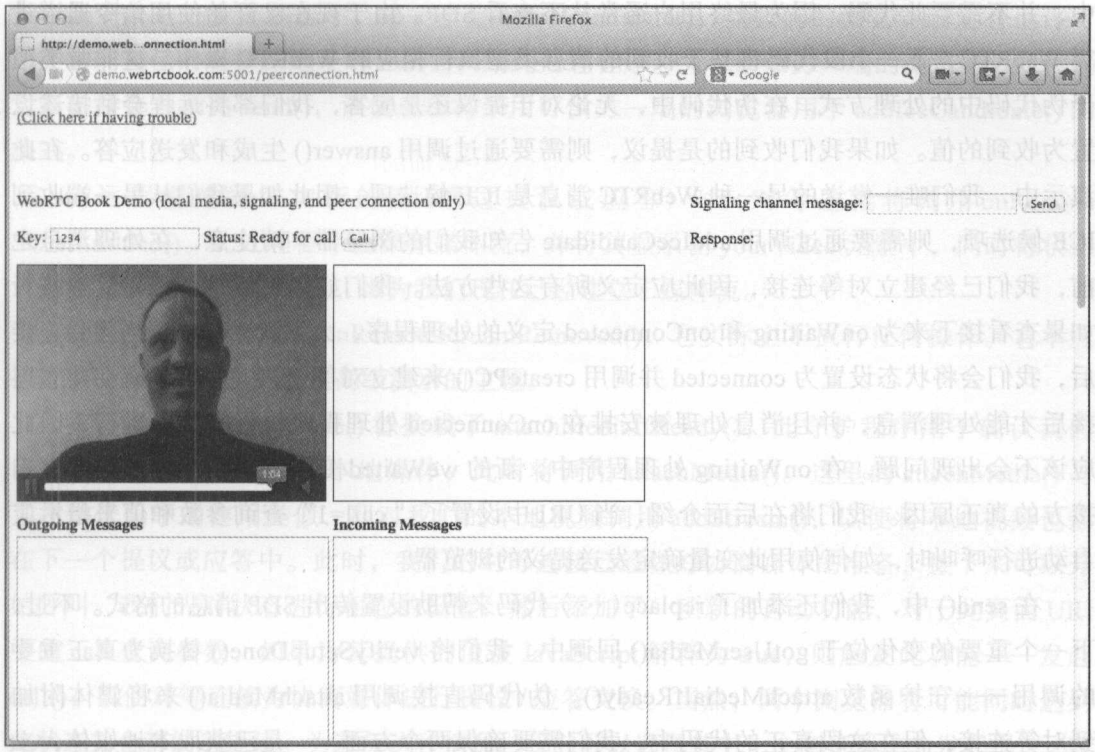


图 6.2　呼叫就绪

现在我们已有大体了解，下面来看 HTML 文件的开头。大家将会注意到的第一个变化是，代码中出现了一些新的变量。暂时只需注意一个变量 pc，此变量将承载 RTCPeerConnection 对象。它实际上替代了 connected 变量，因为一旦连接成功，就会创建对等连接，稍后我们将会看到这一点。因此，此变量拥有既定的值，其作用与我们之前代码迭代中的 connected === true 相同。

虽然页面的 onload 处理程序只增加了一行代码，用于存储对于远程视频的视频元素的引用，但现在这里多了一些重要的注释。在伪代码中，我们先调用 getMedia() 和 createPC()，然后调用 attachMedia()，以此将本地媒体附加到新创建的对等连接中。如前所述，在这段真正的代码中，当两个浏览器之间建立了信令连接（当然，仍通过 Web 服务器）时，将由 connect() 来调用 createPC()，只有当 createPC() 和 getMedia() 都成功执行后，才会调用 attachMedia()，因为要将媒体附加到对等连接，我们既需要建立对等连接，又需要获取媒体！当我们看到执行此功能的代码时，就会更明白这一点。

虽然我们已在第 4 章中介绍了基本 connect() 例程的工作方式，但这次还是针对要通过信令通道中继的提议 / 应答消息添加了一些特定的代码。新增的第一项内容是 replace（…）代码，专门用于改善 SDP 在消息窗口中的视觉显示效果。在大多数 WebRTC 应用程序中，并不需要此代码，因为最终用户通常从不查看 SDP。由于现在已开始使用信令通道进行 WebRTC，下一小段代码将基于收到的消息类型执行相应的 WebRTC 操作。这非常类似于伪代码中的处理方式。在伪代码中，无论对于提议还是应答，我们都将远程会话描述设置为收到的值。如果我们收到的是提议，则需要通过调用 answer() 生成和发送应答。在此演示中，我们唯一发送的另一种 WebRTC 消息是 ICE 候选项，因此如果我们从另一端收到 ICE 候选项，则需要通过调用 addIceCandidate 告知我们的浏览器。请注意，在处理消息之前，我们已经建立对等连接，因此应定义所有这些方法。我们如何判断是否已建立连接？如果查看接下来为 onWaiting 和 onConnected 定义的处理程序，大家就会发现，当建立连接后，我们会将状态设置为 connected 并调用 createPC() 来建立对等连接。由于只有在建立连接后才能处理消息，并且消息处理被安排在 onConnected 处理程序执行完毕后进行，因此应该不会出现问题。在 onWaiting 处理程序中，新的 weWaited 变量可揭示我们监控第一连接方的真正原因。我们将在后面介绍，当 URI 中设置了“ call=1”查询参数和值来指示应自动进行呼叫时，如何使用此变量确定发送提议的浏览器。

在 send() 中，我们还添加了 replace（…）代码来帮助设置传出 SDP 消息的格式。不过，下一个重要的变化位于 gotUserMedia() 回调中。我们将 verifySetupDone() 替换为真正重要的调用——守护函数 attachMediaIfReady()。伪代码直接调用 attachMedia() 来将媒体附加到对等连接，但在这段真正的代码中，我们需要确保两个方面：一是已获取本地媒体；二是已创建对等连接。由于二者异步进行并且都需要很长时间，因此在获取本地媒体后及建立对等连接后，都会调用检查函数 attachMediaIfReady()，一旦二者均已就绪，则可以调用 attachMedia()。

这里的 createPC() 非常类似于伪代码。我们先创建一个新连接，再为其设置各种处理程序。与 getMedia() 一样，我们最后都调用 attachMediaIfReady() 检查函数来收尾。关于在新 RTCPeerConnection() 调用的配置中加入 STUN 和 TURN 服务器 URI 的这段代码，有必要略微解释一下。第一小段代码旨在检查是否设置了 stunuri 和 / 或 turnuri 查询参数。myfalse() 例程处理表示 true 的 JavaScript 值，例如字符串“0”，其方式是将这些值转换为 false，以便查询参数按预期方式发挥作用。接下来，我们使用代码中的 stunuri 和 turnuri 标记来确定是否要为对等连接配置 STUN 和 TURN 服务器。此代码本应简单得多，但 Google Chrome 的 WebRTC 实施方案有一个 Bug：对于你提供的 TURN URI，如果相应的 TURN 服务器仅支持 TURN（而不像许多其他型号一样也支持 STUN），则会忽略来自各 STUN 服务器的所有 STUN 响应，即使是服务器列表中提供的 STUN 服务器，情况也不例外。无论

如何，如果请求的是 STUN 服务器（默认设置），此代码将会添加 STUN 服务器，如果请求的是 TURN 服务器，此代码将会选择仅支持 TURN 的服务器或同时支持 STUN 的 TURN 服务器，具体取决于是否也请求了 STUN 服务器。

接下来是三个处理程序。第一个是 onIceCandidate()，当我们的浏览器确定有也许可用于访问自己的新候选地址时，就会调用此处理程序。为了通知另一端的浏览器，此代码对新的候选项调用 send()，该候选项将采用可供另一端的浏览器用于 addIceCandidate() 的格式。

当远程浏览器添加新的流（通过提议或应答）时，对等连接将调用 onRemoteStreamAdded()。此处理程序会保存媒体流，并将其显示在 yourVideo 元素中，同时将状态设置为“On Call”（通话中），因为我们现在已经建立了媒体流。

最后一个处理程序是 onRemoteStreamRemoved()，它实际上不执行任何操作。它本应执行某项操作，但这属于更高级演示的主题。

我们将 verifySetupDone() 替换成了 attachMediaIfReady()，此守护程序用于确认我们已建立对等连接并获取了本地媒体，此时将调用 attachMedia()。这里的 attachMedia() 与其在伪代码中的作用类似，即对我们的本地视频调用 addStream()，以便将本地视频包括在下一个提议或应答中。此时，我们的对等连接已经做好协商媒体的准备，接下来可以开始呼叫。我们先对状态进行相应的设置，然后添加了一项新的自动功能，对于此页面 URI 中的 call 查询参数，如果为其提供的值被 JavaScript 解释为 true，则触发此功能——发起 call()，以便对等连接为协商媒体进行提议 / 应答交换。当然，两个浏览器有可能同时达到此状态并全都发送提议。为避免这种情况，我们可以选择其中一个浏览器作为发起提议 / 应答交换的一方。由于发送至我们 Node 服务器的所有请求都已自动完成序列化，因此总会有一个浏览器最先进行连接，并从服务器收到 waiting 状态。该浏览器有望已经建立对等连接并准备好本地媒体，因此当状态不是 waiting 的另一个浏览器将其媒体附加到对等连接时，应该就可以开始呼叫了。简而言之，通过让状态不是 waiting 的浏览器调用 call()，确实可以提高双方一切就绪的概率，但我们并不能提供 100% 的保证。下面谈谈 key 和 call 查询参数，其有趣之处在于，两个用户只需加载形如“ http://www.example.org:5001/start.html?key=1234&call=1”的同一 URI，两个应用程序应该就可以自动建立连接——先建立信令通道，再建立对等连接 / 媒体。对于此类 URI，甚至还可以将其添加为书签并以该用户的姓名命名，或者存储到与该用户的姓名相对应的地址簿条目中。

接下来，最后一节新代码负责处理呼叫和应答（提议和应答）。call() 调用 createOffer()，而 answer() 调用 createAnswer()。在这两种情况下，一旦会话描述准备就绪，就会将其提供给 gotDescription 回调，该回调会将其设置为我们的本地描述，再发送给另一端的浏览器。当本地描述和远程描述都设置完毕后，两个浏览器即可启动媒体并建立“通话”（如图 6.3 所示）！

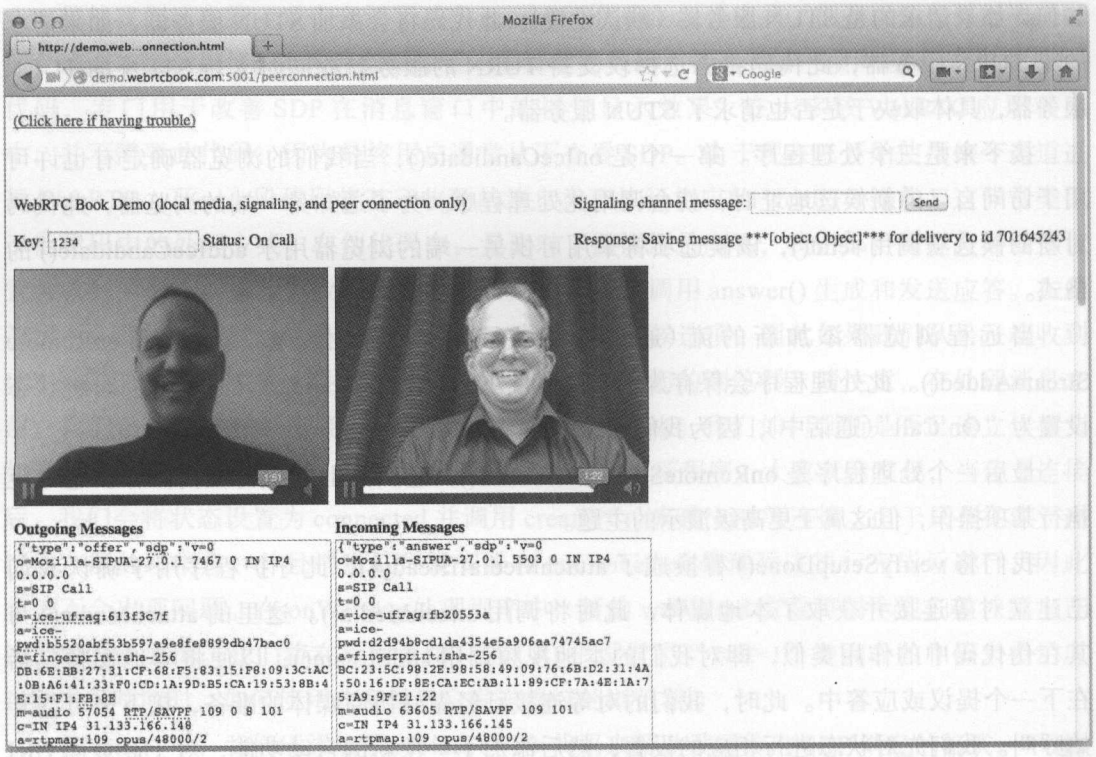
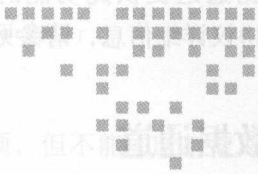


图 6.3 通话成功建立

请注意，此代码示例只提供了一个提议和一个应答。在实际应用程序中，应利用negotiationneeded来处理需要重新协商媒体的情形。



第 7 章 Chapter 7

数据通道

WebRTC 数据通道是在浏览器之间建立的一种非媒体交互连接。它为 Web 开发人员提供了一种灵活且可配置的通道，用于绕过服务器来直接交换数据。对于某些应用程序，借助采用 WebSocket 连接或 HTTP 消息的服务器传递少量数据就足以满足需要。相比之下，数据通道支持流量大、延迟低的连接，既稳定可靠，又不失灵活性。本章将介绍数据通道、JavaScript API 以及底层协议。

7.1 数据通道简介

虽然有关 WebRTC 的宣传主要侧重于它对于对等媒体的支持，但其原设计师一直都希望它也支持实时数据传输。

数据通道模型基于 WebSocket 建立，具有简单且可设置的 send 方法和 onmessage 处理程序。数据通道的创建非常简单，下一节将对此进行介绍。不过，与媒体不同的是，单个对等连接中的多个数据通道全都使用同一底层流。在实际中，这意味着只需进行一次提议 / 应答协商即可建立首个数据通道，之后将自动根据用于该数据通道的协议协商所有新的数据通道，因此无须在 JavaScript 级别进行额外的提议 / 应答交换即可添加更多数据通道。

数据通道的目标使用情形之一是提供实时游戏状态更新。经过不断讨论，人们发现既需要有保证（可靠）的传输（即使消息因此而延迟也不例外），又需要不太可靠但迅速的传输。前者适用于传输关键事件，例如“你击中我了”，而后者则适用于传输定时发来的位置更新等消息。

有关为数据通道提供此功能的流控制传输协议（Stream Control Transport Protocol，SCTP）的底层协议详细信息，请参见 10.2.12 节、11.3.6 节和 11.3.7 节。

7.2 使用数据通道

数据通道 API 是对等连接 API 的一部分，因此只有在创建 RTCPeerConnection 实例后才能创建数据通道。

```
pc = new RTCPeerConnection();
dc = pc.createDataChannel("dc1");
```

这是创建数据通道的最简便方式。在此示例中，新数据通道的标签将为 dc1。前面提到，此类数据通道只有第一个需要在应用程序级别进行提议 / 应答协商。利用上面显示的简单语法创建数据通道时，将导致对等端收到 RTCDataChannelEvent，通过为对等连接设置 ondatachannel 处理程序，可在对等端处理该事件：

```
pc = new RTCPeerConnection();
pc.ondatachannel = function(e) {
  dc = e.channel;
};
```

此时，两个对等端之间已建立双向数据通道。无论数据通道通过 createDataChannel() 创建还是通过 ondatachannel 返回，现在都可以通过 send() 方法和 onmessage 处理程序来使用它：

```
dc.send("I can send a text string");
dc.send(new Blob(["I can send blobs"], {type: "text/plain"}));
dc.send(new arrayBuffer(32));      // 发送 arrayBuffer
dc.send(new uInt8Array([1,2,3])); // 还发送 arrayBufferViews
dc.onmessage = function(e) {
  console.log("Received message" + e.data);
}
```

数据通道有多种配置选项。这些选项只能在数据通道创建之后使用，因为需要在对等端之间围绕它们进行协商。所有这些选项都作为 RTCDataChannelInit 对象的属性包括在内。你可能使用的属性分为两类：可靠型和创建型。在默认情况下，将创建可靠的数据通道，这意味着如果有任何数据包丢失，底层传输机制都会通过重新传输自动找回这些数据包。当然，如果连接十分糟糕，没有数据包能够送达，则意味着你根本没有建立连接。假设你已经建立连接，并且大多数数据包都能够送达，这时你可以通过两个配置参数控制重新传输的级别：

```
// 限制通道在第一次数据传输失败时重新传输数据的次数
dc = pc.createDataChannel("", {maxRetransmits:3});
```

```
// 限制通道允许重试操作持续的毫秒数
dc = pc.createDataChannel("", {maxRetransmitTime:30});

// 注意：这两个属性互相排斥
```

对于同一数据通道，你可以指定以上任一配置选项，但不能同时指定两者，因为它们互相排斥。何时你可能希望降低数据通道的可靠性呢？这包括两种情形：一是你认为速度更重要；二是丢失数据并不要紧，这可能是因为接连发送的值会不断取代之前的值。

由于每次发送都通过网络独立进行，因此不同的消息可能采用不同的路径，导致最终的送达顺序不同于最初的发送顺序。在默认情况下，WebRTC 数据通道将根据需要进行等待，直至这些消息按原顺序组合起来。例如，如果消息 1 发送失败并需要重试多次，则其成功送达的时间可能晚于消息 2。在默认情况下，用户代理将等待成功收到消息 1，再将消息 1 和消息 2 释放给 onmessage 处理程序。在此方面，还有一个用于施加控制的配置属性：

```
// 允许在消息送达时接收它们，即使其顺序混乱也不例外
dc = pc.createDataChannel("", {ordered: false});
```

通常，这可能有助于更快地传输至你的应用程序。当消息彼此足够独立，而顺序并不重要时，就适合采取这种处理方式。

由于远程对等端并不创建数据通道，而只是接收有关数据通道已创建的通知，因此你此时可能想知道，远程对等端如何为以上设置建立不同于发起对等端的值。当然，远程对等端也可以调用 createDataChannel() 并创建具有不同属性的另一个双向通道，但这似乎有些浪费。幸运的是，你可以通过一些配置选项来创建不同类型的数据通道，并单独配置每个方向：

```
// 本地对等端：创建数据通道的前半部分
properties = {negotiated: true, id:1, maxRetransmits:3});
dc = pc.createDataChannel("", properties);

// 远程对等端：创建数据通道的后半部分
properties = {negotiated: true, id:1, maxRetransmits:6});
dc = pc.createDataChannel("", properties);
```

现在，我们创建了一个双向数据通道，它允许本地浏览器向远程浏览器重新传输 3 次，并允许远程浏览器向我们重新传输 6 次。这里有两个关键点：一是 negotiated 属性，此属性必须设置为 true；二是 id 属性，两个对等端必须将此属性设置为同一个值，才能将两个单向数据通道关联起来。请注意，如果未设置 id 属性，浏览器将为其选择一个值。因此，在创建通道的前半部分时，应用程序可能会让浏览器选择 id，并通过信令通道将该 id 发送至对等端，然后让对等端使用该 id 创建数据通道。

有关可设置和检查的各种属性的详细信息，请参见 8.3.1.2 节。

7.3 可运行的数据通道代码示例

现在，我们可以在本书一直扩充的代码示例中添加最后一部分——数据通道。同样，服务器和信令代码保持不变。唯一的新代码位于该 HTML 应用程序的 JavaScript 中。下一节将显示完整的代码，我们仍然是先重点查看新增的代码，再解释它们。你可能会惊讶地发现，所需的新代码其实很少。

客户端 WebRTC 应用程序

```
<!--
// 版权所有 2013-2014 Digital Codex LLC 你可以将此代码用于自身学习。如果你基本按照原样使用此代码，或者基于此代码开发了新的代码，
// 请勿标榜你的代码先于此代码问世
// 使用此代码的风险完全由你自己承担
// 如果你在工作中过度依赖此代码，则并非明智之举，我们对此概不负责
-->

<html>
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
  <style>
    video {
      width:  320px;
      height:  240px;
      border:  1px solid black;
    }
    div {
      display:  inline-block;
    }
  </style>
</head>
<body>

<!-- 此空白脚本部分专为查询参数预留 -->
<script></script>

<!-- 加载 polyfill，先加载本地副本以进行本地测试 -->
<script src="extra/adapter.js" type="text/javascript"></script>
<script
  src="https://webrtc.github.io/adapter/adapter.js"
  type="text/javascript"></script>

<!-- 加载基于 XHR 的信令通道，以基于密钥定向连接 -->
<script src="clientXHRSignalingChannel.js"
        type="text/javascript"></script>

<script>
```

```
var signalingChannel, key, id,
    haveLocalMedia = false,
    weWaited = false,
    myVideoStream, myVideo,
    yourVideoStream, yourVideo,
    doNothing = function() {},
    pc, dc, data = {},
    constraints = {mandatory: {
                      OfferToReceiveAudio: true,
                      OfferToReceiveVideo: true}};

////////////////////////////
// 这是主例程
////////////////////////////

// 由此开始获取本地媒体
// 此外，它还可以自动启动信令通道
window.onload = function () {

  // 如果URI中提供了密钥，则自动连接信令通道
  if (queryparams && queryparams['key']) {
    document.getElementById("key").value = queryparams['key'];
    connect();
  }

  myVideo = document.getElementById("myVideo");
  yourVideo = document.getElementById("yourVideo");

  getMedia();

  // 如果建立连接，connect()将调用createPC()
  // 如果createPC()和getMedia()成功执行，将调用attachMedia()
};


////////////////////
// 接下来这一节代码用于建立信令通道。
////////////////////

// 此例程会连接至Web服务器并建立信令通道
// 当加载文档或用户单击"Connect"（连接）按钮时，将自动调用此例程。
function connect() {
  var errorCB, scHandlers, handleMsg;

  // 首先，获取用于连接的密钥
  key = document.getElementById("key").value;

  // 此处理程序用于处理通过信令通道收到的所有消息。
  handleMsg = function (msg) {
    // 首先，我们整理消息并将其发布到屏幕上
```

```
    var msgE = document.getElementById("inmessages");
    var msgString = JSON.stringify(msg).replace(/\\r\\n/g,'\n');
    msgE.value = msgString + "\n" + msgE.value;

    // 随后，我们基于消息的类型执行操作
    if (msg.type === "offer") {
      pc.setRemoteDescription(new RTCSessionDescription(msg));
      answer();
    } else if (msg.type === "answer") {
      pc.setRemoteDescription(new RTCSessionDescription(msg));
    } else if (msg.type === "candidate") {
      pc.addIceCandidate(
        new RTCIceCandidate({sdpMLineIndex:msg.mlineindex,
                             candidate:msg.candidate}));
    }
  };

  // 用于信令通道的处理程序
  scHandlers = {
    'onWaiting' : function () {
      setStatus("Waiting");
      // 后面将使用 weWaited 执行自动调用
      weWaited = true;
    },
    'onConnected': function () {
      setStatus("Connected");
      // 由于我们已成功连接，因此建立 RTC 对等连接
      createPC();
    },
    'onMessage': handleMsg
  };

  // 最后，创建信令通道
  signalingChannel = createSignalingChannel(key, scHandlers);
  errorCB = function (msg) {
    document.getElementById("response").innerHTML = msg;
  };

  // 进行连接
  signalingChannel.connect(errorCB);
}


// 此例程通过信令通道发送消息
// 其方式有二：一是执行显式调用；二是通过用户单击 "Send"（发送）按钮。
function send(msg) {
  var handler = function (res) {
    document.getElementById("response").innerHTML = res;
    return;
  },
```

```
  // 如果没有传入，则获取消息
  msg = msg || document.getElementById("message").value;

  // 整理消息并发布到屏幕上
  msgE = document.getElementById("outmessages");
  var msgString = JSON.stringify(msg).replace(/\\r\\n/g,'\n');
  msgE.value = msgString + "\n" + msgE.value;

  // 同时通过信令通道发送
  signalingChannel.send(msg, handler);
}

////////////////////////////
// 接下来这一节代码用于获取本地媒体
////////////////////////////

function getMedia() {
  getUserMedia({"audio":true, "video":true},
               gotUserMedia, didntGetUserMedia);
}

function gotUserMedia(stream) {
  myVideoStream = stream;
  haveLocalMedia = true;

  // 向我显示我的本地视频
  attachMediaStream(myVideo, myVideoStream);
  // 等待 RTCPeerConnection 创建完毕
  attachMediaIfReady();
}

function didntGetUserMedia() {
  console.log("couldn't get video");
}

////////////////////////////
// 接下来这一节代码用于建立 RTC 对等连接
////////////////////////////

function createPC() {
  var stunuri = true,
      turnuri = false,
      myfalse = function(v) {
                  return ((v==="0")||(v==="false")||(!v)); },
      config = new Array();

  // 基于各个查询参数调整配置字符串
  if (queryparams) {
    if ('stunuri' in queryparams) {
      stunuri = !myfalse(queryparams['stunuri']);
```

```
    }
    if ('turnuri' in queryparams) {
      turnuri = !myfalse(queryparams['turnuri']);
    };
  };

  if (stunuri) {
    // 这是Google的一台公共STUN服务器
    config.push({"url":"stun:stun.l.google.com:19302"});
  }
  if (turnuri) {
    if (stunuri) {
      // 这里不能使用仅支持TURN的TURN服务器,
      // 因为Chrome中存在一个Bug，可导致STUN服务器的响应遭到忽略,
      // 因此我们使用同时具备STUN功能的TURN服务器
      config.push({"url":"turn:user@turn.webrtcbook.com",
                   "credential":"test"});
    } else {
      // 这是我们仅支持TURN的TURN服务器
      config.push({"url":"turn:user@turn-only.webrtcbook.com",
                   "credential":"test"});
    }
  }
  console.log("config = " + JSON.stringify(config));

  pc = new RTCPeerConnection({iceServers:config});
  pc.onicecandidate = onIceCandidate;
  pc.onaddstream = onRemoteStreamAdded;
  pc.onremovestream = onRemoteStreamRemoved;
  pc.ondatachannel = onDataChannelAdded;

  // 等待本地媒体准备就绪
  attachMediaIfReady();
}

// 如果我们的浏览器有另一个候选项，则将其发送给对等端
function onIceCandidate(e) {
  if (e.candidate) {
    send({type:  'candidate',
          mlineindex:  e.candidate.sdpMLineIndex,
          candidate:  e.candidate.candidate});
  }
}

// 如果我们的浏览器检测到另一端添加了媒体流，则将其显示在屏幕上
function onRemoteStreamAdded(e) {
  yourVideoStream = e.stream;
  attachMediaStream(yourVideo, yourVideoStream);
  setStatus("On call");
}
```

```
// 没错，如果远程端移除该流，我们不执行任何操作。
// 毕竟这是一个 * 简单 * 的演示。
function onRemoteStreamRemoved(e) {}

// 如果我们的浏览器检测到另一端添加了数据通道，
// 则保存该数据通道、设置处理程序并发送欢迎消息
function onDataChannelAdded(e) {
  dc = e.channel;
  setupDataHandlers();
  sendChat("hello");
}

// 设置数据通道消息处理程序
function setupDataHandlers() {
  data.send = function(msg) {
    msg = JSON.stringify(msg);
    console.log("sending " + msg + " over data channel");
    dc.send(msg);
  }
  dc.onmessage = function(e) {
    var msg = JSON.parse(e.data),
        cb = document.getElementById("chatbox"),
        rtt = document.getElementById("rtt");

    if (msg.rtt) {
      // 如果是实时文本（基于 keypress）消息，则将其显示在实时窗口中
      console.log("received rtt of '" + msg.rtt + "'");
      rtt.value = msg.rtt;
      msg = msg.rtt;
    } else if (msg.chat) {
      // 如果是完整的消息，
      // 则将其显示在聊天窗口中，
      // 重置实时窗口，并强制聊天窗口滚动到最后一行
      console.log("received chat of '" + msg.chat + "'");
      cb.value += "<- " + msg.chat + "\n";
      rtt.value = "";
      cb.scrollTop = cb.scrollHeight;
      msg = msg.chat;
    } else {
      console.log("received " + msg + "on data channel");
    }
  };
}

// 发送实时文本。基本而言，对于每个 keyup 事件，
// 我们都在实时消息允许的范围内发送整个字符串，
// 以便在每次发生 keyup 事件时都显示相应的消息
function sendRtt() {
  var msg = document.getElementById("chat").value;
  data.send({'rtt':msg});
```

```
}

// 发送常规聊天消息。
// 当发生 enter keyup 事件时，就会这样处理，该事件意味着远程用户已输入完一行内容
// 它还用于发送我们的初始 hello 消息。
function sendChat(msg) {
  var cb = document.getElementById("chatbox"),
      c = document.getElementById("chat");

  // 在本地显示消息、发送消息并强制聊天窗口滚动到最后一行
  msg = msg || c.value;
  console.log("sendChat(" + msg + ")");
  cb.value += "->" + msg + "\n";
  data.send({'chat':msg});
  c.value = '';
  cb.scrollTop = cb.scrollHeight;
}


////////////////////////////////////
// 接下来这一节代码用于将本地媒体附加到对等连接。
////////////////////////////////////

// 此守护例程实际上用于对两项异步活动的完成时间进行同步：
// 一是创建对等连接；二是获取本地媒体。
function attachMediaIfReady() {
  // 如果 RTCPeerConnection 已经就绪，并且我们已获得本地媒体，
  // 则继续处理。
  if (pc && haveLocalMedia) {attachMedia();}
}

// 此例程将我们的本地媒体流添加至对等连接
// 请注意，这不会导致任何媒体开始流动
// 其作用只是指示浏览器在下一个 SDP 描述中加入此流
function attachMedia() {
  pc.addStream(myVideoStream);
  setStatus("Ready for call");

  // 如果 URI 中 call 参数的值表示 true，则自动执行调用，
  // 但还要确保我们已完成连接之前的所有步骤（提高两端已一切就绪的概率）
  if (queryparams && queryparams['call'] && !weWaited) {
    call();
  }

}

////////////////////////////
// 接下来这一节代码用于呼叫和应答
////////////////////////////

// 这会创建数据通道并为提议生成会话描述
```

```
function call() {
  dc = pc.createDataChannel('chat');
  setupDataHandlers();

  pc.createOffer(gotDescription, doNothing, constraints);
}

// 以下代码为应答生成会话描述
function answer() {
  pc.createAnswer(gotDescription, doNothing, constraints);
}

// 无论在哪一种情况下，一旦我们获取会话描述，
// 就指示我们的浏览器将其用作本地描述，
// 然后将其发送给另一端的浏览器
// 只有先设置了本地描述，浏览器才能发送媒体并准备从另一端接收媒体
function gotDescription(localDesc) {
  pc.setLocalDescription(localDesc);
  send(localDesc);
}

////////////////////////////////////
// 这一节代码用来基于应用程序的进度更改 UI。
////////////////////////////////////

// 此函数将通过隐藏、显示和填充各种 UI 元素，
// 让用户大体了解浏览器在建立信令通道、
// 获取本地媒体、创建对等连接，
// 以及实际连接媒体（呼叫）方面的进度。
function setStatus(str) {
  var statuslineE = document.getElementById("statusline"),
      statusE = document.getElementById("status"),
      sendE = document.getElementById("send"),
      connectE = document.getElementById("connect"),
      callE = document.getElementById("call"),
      scMessageE = document.getElementById("scMessage");

  switch (str) {
    case 'Waiting':
      statuslineE.style.display = "inline";
      statusE.innerHTML =
        "Waiting for peer signaling connection";
      sendE.style.display = "none";
      connectE.style.display = "none";
      break;
    case 'Connected':
      statuslineE.style.display = "inline";
      statusE.innerHTML =
        "Peer signaling connected, waiting for local media";
      sendE.style.display = "inline";
```

```
        connectE.style.display = "none";
        scMessageE.style.display = "inline-block";
        break;
      case 'Ready for call':
        statusE.innerHTML = "Ready for call";
        callE.style.display = "inline";
        break;
      case 'On call':
        statusE.innerHTML = "On call";
        callE.style.display = "none";
        break;
      default:
    }
  }

  </script>

  <div id="setup">
    <p>WebRTC Book Demo (local media, signaling, peer connection, and data
channel)</p>
    <p>Key:
      <input type="text" name="key" id="key"
             onkeyup="if (event.keyCode == 13) {
                      connect(); return false;}"/>
      <button id="connect" onclick="connect()">Connect</button>
      <span id="statusline" style="display:none">Status:
        <span id="status">Disconnected</span>
      </span>
      <button id="call" style="display:none"
              onclick = "call()">Call</button>
    </p>
  </div>

  <div id="scMessage" style="float:right;display:none">
    <p>Signaling channel message:
      <input type="text" width="100%" name="message" id="message"
             onkeyup="if (event.keyCode == 13) {
                      send(); return false;}"/>
      <button id="send" style="display:none"
              onclick="send()">Send</button>
    </p>

    <p>Response:  <span id="response"></span></p>
  </div>

  <br/>
```

```
<div style="width:30%;vertical-align:top">
  <div>
    <video id="myVideo" autoplay="autoplay" controls
           muted="true"/>
  </div>
  <p><b>Outgoing Messages</b>
    <br/>
    <textarea id="outmessages" rows="100"
              style="width:100%"></textarea>
  </p>
</div>

<div style="width:30%;vertical-align:top">
  <textarea id="chatbox" rows="10" style="width:100%"></textarea>
  <p style="width:100%"><b>Real-time:</b>
    <textarea id="rtt" rows="2" style="width:100%"></textarea>
  </p>
  <p style="width:100%"><b>Chat message:</b>
    <input type="text" style="width:100%" name="chat" id="chat"
           onkeyup="sendRtt();
                    if (event.keyCode == 13) {
                      sendChat(); return false;}"/>
  </p>
</div>

<div style="width:30%;vertical-align:top">
  <div>
    <video id="yourVideo" autoplay="autoplay" controls />
  </div>
  <p><b>Incoming Messages</b>
    <br/>
    <textarea id="inmessages" rows="100"
              style="width:100%"></textarea>
  </p>
</div>

</body>
</html>
```

现在，文件末尾新增了一节代码，专门用于呈现数据通道的用户界面（如图 7.1 所示）。它包含聊天历史记录窗口、实时文本窗口和消息输入字段。该输入字段用于输入要发送至另一端浏览器的消息。每个字符在输入后将被立即发送至另一端的浏览器，并显示在实时文本窗口中（实际上，发送的内容与输入的内容略有不同，稍后我们将对此进行介绍）。当你按 Enter 键时，整个消息将被发送至另一端的浏览器，并显示在其顶部的聊天历史记录窗口中。我们将会看到，新增的大部分 JavaScript 代码用于显示和格式设置，因为数据通道本身的用法非常简单。

图 7.1 数据通道演示

下面我们来看一遍 JavaScript 代码。新增的第一小段 JavaScript 代码用于声明两个新变量：一是 dc，用于承载指向实际数据通道的指针；二是 data，用于包装数据通道的方法，并能够为要发送和显示的消息设置格式。接下来这行新代码用于设置 ondatachannel 处理程序，当数据通道由对等端创建时，就需要此代码。下面这些代码仍是定义部分。此代码将对等端创建的通道保存到 dc 变量中，并为数据通道设置处理程序，然后向对等端发送第一条消息。

接下来，setupDataHandlers() 设置 data 变量的 send() 方法。此方法会收集所有消息对象、将其转换为 JSON 并进行发送。其方便之处在于，它提供全面的支持，既支持简单的字符串，也支持任意数据对象。我们正是利用这一点来通过数据通道发送各种消息的。下面，我们暂时跳过 dc.onmessage 处理程序，先来看 data.send() 的使用方式。最常见的消息是通过 sendRtt() 发送的消息。每当聊天输入字段中出现 keyup 事件时，我们都捕获该输入字段中的整个字符串，并将其作为一个新对象的 rtt 属性值来通过数据通道发送。虽然发送整个字符串会额外增加几字节的数据发送量，但有助于简化编程，因为我们不需要跟踪字符的位置、删除操作等因素。此外，当某一条实时消息丢失时，这种处理还可以实现自我纠正。

我们发送的另一种消息是常规聊天消息，此类消息通过 sendChat() 发送。如果存在传

入的 msg，此例程将使用该 msg，否则将使用输入字段中的值。在将数据写入控制台后，它会将消息附加到聊天历史记录窗口，并用箭头指示它是来自我们的传出消息。随后，它将消息作为一个新对象的 chat 属性通过数据通道发送，接着清空聊天输入字段，最后强制聊天窗口滚动到最后一行，以确保我们能够看到该消息。图 7.2 显示了一个实时文本和聊天历史记录的示例。

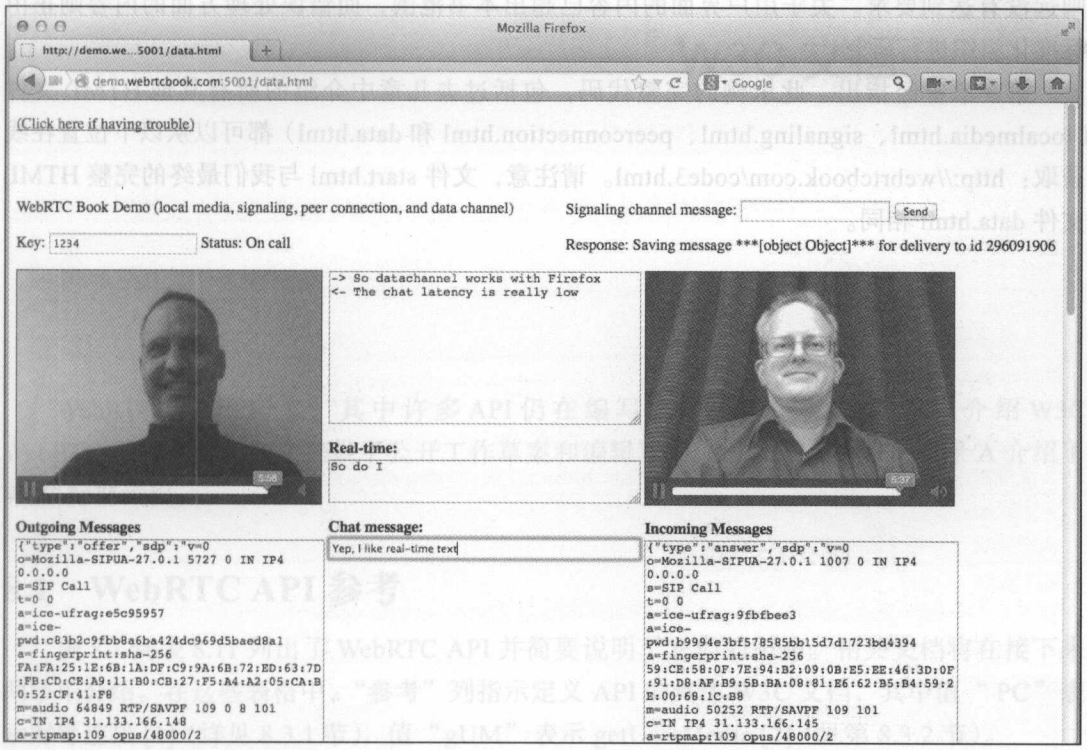


图 7.2 通过数据通道发送聊天和实时文本

现在，我们回过头来看一下数据通道的 onmessage 处理程序。我们先分析 JSON 字符串，将其还原为一个对象。此处理程序会检查 rtt 和 chat 属性。如果存在前者，则记录消息并将其保存到实时窗口中。如果存在后者，则记录消息将其附加到聊天历史记录窗口并用箭头指示其为传入的消息，再清空实时窗口，然后强制聊天窗口滚动到最后一行（与处理传出消息时一样）。如果 rtt 和 chat 都不存在，则将其视为一个简单的字符串并进行记录。

新增的最后一小段代码用于创建数据通道。不知大家是否记得，到目前为止，我们见过的发起 setupDataHandlers() 的唯一代码是在对等端创建数据通道时激活的代码。我们当前要讨论的这段代码就执行这一功能。它极为简单，只包含对 createDataChannel() 的调用。

这是我们在示例中所做的最后一项扩充。请注意数据通道的建立是多么简单。实际上，创建和使用数据通道本身几乎简单至极；新增的大部分代码都是设置格式、进行分析，或

基于收到的消息执行操作所必需的。由于数据通道的建立非常简单，而且数据通道消息送达远程对等端的速度可能快于通过信令通道中继的消息，因此在实际应用程序中，比较合理的处理方式可能是先建立数据通道，再照第 4.3.8 节所述使用该数据通道作为完成所有其他设置的信令通道。

现在，从功能上讲，此应用程序已经相当完整，但从用户界面和错误处理角度而言，则远没有达到要求。关于用户界面的内容已超出本书范围，而错误处理方面的内容则正由标准化组织进行研究！

第 3 章曾经指出，此示例的完整代码，包括过去几章中介绍的所有其他 HTML 文件（localmedia.html、signaling.html、peerconnection.html 和 data.html）都可以从以下位置在线获取：http://webrtcbook.com/code3.html。请注意，文件 start.html 与我们最终的完整 HTML 文件 data.html 相同。

第 8 章 Chapter 8

W3C 文档

WebRTC 由 API 定义，其中许多 API 仍在编写之中。以下几节内容将介绍 W3C WebRTC 标准文档。文中提供了公开工作草案和编辑草案，以便读者参考。附录 A 介绍了 W3C 标准流程。

8.1 WebRTC API 参考

表 8.1 到表 8.11 列出了 WebRTC API 并简要说明了它们的用途。相关文档将在接下来两节中介绍。在这些表格中，“参考”列指示定义 API 的具体 W3C 文档，其中值“PC”表示对等连接 [1]（详见 8.3.1 节），值“gUM”表示 getUserMedia [2]（见第 8.3.2 节）。

表 8.1 WebRTC API 接口摘要（PC= 对等连接 [1]；gUM=getUserMedia [2]）

接口和说明	参考
AudioMediaStreamTrack MediaStreamTrack 的子类，只能承载 sourceType 为“音频”的媒体。	gUM
MediaStream 表示 MediaStreamTrack 的集合，目前仅包含音频和视频。	gUM
MediaStreamEvent 返回由远程对等端添加或删除的 MediaStream。 由 onaddstream 或 onremovestream 处理。	PC
MediaStreamTrack 表示媒体源的单个轨道。请注意，每个轨道可包含多个通道，例如编码为单个轨道的 6 声道环绕声源。此外，每个轨道只能包含一种媒体，而不论它有多少个通道。	gUM
MediaStreamTrackEvent 在添加或删除轨道时返回一个 MediaStreamTrack。 由 onaddtrack 或 onremovetrack 处理。	gUM

（续）

接口和说明	参考
NavigatorUserMedia 所有 Web 浏览器中预先存在的一个接口。	gUM
Constrainable 添加用于操作约束、功能和设置的方法和属性。由 MediaStreamTrack 实现。	PC
MediaError 表示调用 NavigatorUserMedia.getUserMedia() 时返回的错误。	gUM
MediaErrorEvent 返回一个 MediaError 对象，其中包含错误名称和消息；如果错误名称为“CONSTRAINT_NOT_SATISFIED”，则还会一并返回未满足的强制性约束。后一种情形可由 onoverconstrained 处理。	gUM
RTCDataChannel 表示 RTCPeerConnection 上可传输任意应用程序数据的通道。	PC
RTCDataChannelEvent 当远程对等端创建了 RTCDataChannel 时返回 RTCDataChannel。由 ondatachannel 处理。	PC
RTCIceCandidate ICE 候选项的容器。	PC
RTCPeerConnection 表示两个对等端之间的 WebRTC 连接。	PC
RTCPeerConnectionIceEvent 当浏览器识别到 RTCIceCandidate 时返回 RTCIceCandidate。由 onicecandidate 处理。	PC
RTCSessionDescription SDP 提议、应答或临时应答 (pranswer) 的容器。	PC
RTCSDPError 包含 setLocalDescription() 或 setRemoteDescription() 因给定 RTCSessionDescription 中的问题而失败时所发生的第一个错误的 SDP 行号。	PC
RTCDTMFSender 用于将 DTMF 插入音频轨道的容器和控制对象。	PC
RTCDTMFToneChangeEvent 返回刚刚开始发送的 DTMF 提示音。由 ontonechange 处理。	PC

表 8.2 WebRTC RTCPeerConnection API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

RTCPeerConnection API 和说明	类型	参考
RTCPeerConnection 表示两个对等端之间的 WebRTC 连接。	接口	PC
新的 RTCPeerConnection(configuration) 使用给定的 STUN 和 TURN 服务器信息创建一个新的 RTCPeerConnection 对象。configuration 参数的类型为 RTCConfiguration。	构造函数	PC
RTCPeerConnection.close() 关闭 RTCPeerConnection，实际上会删除所有附加的流并关闭所有附加的 RTCDataChannel。	方法	PC
RTCPeerConnectionErrorCallback 可由应用程序设置为一个函数 / 方法，该函数 / 方法将错误信息的 DOMString 用作参数。由 RTCPeerConnection.createOffer()、RTCPeerConnection.createAnswer()、RTCPeerConnection.setLocalDescription()、RTCPeerConnection.setRemoteDescription() 和 RTCPeerConnection.getStats() 使 用。	回调	PC

表 8.3 WebRTC SDP 处理 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

SDP 处理 API 和说明	类型	参考
RTCSessionDescription SDP 提议、应答或临时应答（pranswer）的容器。	接口	PC
新的 RTCSessionDescription（descriptionInitDict） 创建一个新的 RTCSessionDescription 对象。descriptionInitDict 参数的类型为 RTCSessionDescriptionInit。	构造函数	PC
RTCSessionDescription.type 指示会话描述是提议、应答还是临时应答。	属性	PC
RTCSessionDescription.sdp 一个字符串，表示会话描述的 SDP。	属性	PC
RTCSessionDescriptionInit 会话描述的容器，用于初始化 RTCSessionDescription 对象。	字典	PC
RTCSessionDescriptionInit.type 指示会话描述是提议、应答还是临时应答。	属性	PC
RTCSessionDescriptionInit.sdp 一个字符串，表示会话描述的 SDP。	属性	PC
RTCSessionDescriptionCallback 可由应用程序设置为一个函数 / 方法，该函数 / 方法接受 RTCSessionDescription 作为参数。由 RTCPeerConnection.createOffer() 和 RTCPeerConnection.createAnswer() 使用。	回调	PC
RTCSDPError 包含 setLocalDescription() 或 setRemoteDescription() 因给定 RTCSessionDescription 中的问题而失败时所发生的第一个错误的 SDP 行号。	接口	PC
RTCSDPError.sdpLineNumber setLocalDescription() 或 setRemoteDescription() 因给定 RTCSessionDescription 中的问题而失败时所发生的第一个错误的 SDP 行号。	属性	PC
RTCIceCandidate.sdpMid 与此候选项关联的 m 行的媒体流标识符。	属性	PC
RTCIceCandidate.sdpMLineIndex 与此候选项关联的 m 行的索引（从零开始）。	属性	PC
RTCPeerConnection.createOffer() 使用表示一整套可用本地媒体流、编解码器选项、ICE 候选项等的 SDP，为提议创建 RTCSessionDescription。	方法	PC
RTCPeerConnection.createAnswer() 使用表示一套相应的可用本地媒体流、编解码器选项、ICE 候选项等的 SDP，为应答创建 RTCSessionDescription。	方法	PC
RTCPeerConnection.setLocalDescription() 将给定的 RTCSessionDescription 对象记录为当前本地描述。如果该对象为最终应答，则媒体将更改 / 开始流动。	方法	PC
RTCPeerConnection.setRemoteDescription() 将给定的 RTCSessionDescription 对象记录为当前远程描述。	方法	PC
RTCPeerConnection.localDescription 表示当前活动的本地描述（SDP）的 RTCSessionDescription。	属性	PC

（续）

SDP 处理 API 和说明	类型	参考
RTCPeerConnection.remoteDescription 表示当前活动的远程描述（SDP）的 RTCSessionDescription。	属性	PC
RTCPeerConnection.signalingState 承载 RTCPeerConnection 上 SDP 交换的状态：stable、have-local-offer、have-remote-offer、have-local-pranswer、have- remote-pranswer 和 closed。	属性	PC
RTCPeerConnection.onnegotiationneeded 可由应用程序设置为一个函数 / 方法，每当 RTCPeerConnection 的本地端或远程端更改将导致需要重新协商的 SDP 更改时，都调用该函数 / 方法。	属性	PC
RTCPeerConnection.onsignalingstatechange 可由应用程序设置为一个函数 / 方法，每当 RTCPeerConnection.signalingState 发生更改时，都调用该函数 / 方法。	属性	PC

表 8.4 WebRTC ICE 处理 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

ICE 处理 API 和说明	类型	参考
RTCIceCandidate ICE 候选项的容器。	接口	PC
新的 RTCIceCandidate(candidateInitDict) 基于输入参数创建一个新的 RTCIceCandidate 对象。此参数的类型为 RTCIceCandidateInit。	构造函数	PC
RTCIceCandidate.candidate 一个字符串，表示 ICE 候选项。	属性	PC
RTCIceCandidate.sdpMid 与此候选项关联的 m 行的媒体流标识符。	属性	PC
RTCIceCandidate.sdpMLineIndex 与此候选项关联的 m 行的索引（从零开始）。	属性	PC
RTCIceCandidateInit 用于初始化 RTCIceCandidate 对象的 ICE 服务器 URL 的容器。	字典	PC
RTCIceCandidateInit.candidate 一个字符串，表示 ICE 候选项。	属性	PC
RTCIceCandidateInit.sdpMid 与此候选项关联的 m 行的媒体流标识符。	属性	PC
RTCIceCandidateInit.sdpMLineIndex 与此候选项关联的 m 行的索引（从零开始）。	属性	PC
RTCIceServer ICE 服务器 URL 的容器。	字典	PC
RTCIceServer.urls STUN 和 / 或 TURN 服务器的一个或多个 URL。	属性	PC
RTCIceServer.username 当 RTCIceServer.urls 为 TURN 服务器的 URL 时使用的用户名。	属性	PC
RTCIceServer.credential 当 RTCIceServer.urls 为 TURN 服务器的 URL 时使用的凭据（例如密码）。	属性	PC
RTCConfiguration 包含 ICE 服务器对象的数组。	字典	PC

（续）

ICE 处理 API 和说明	类型	参考
RTCConfiguration.iceServers RTCIceServer 对象的数组。	属性	PC
RTCConfiguration.iceTransports 指示要使用的 ICE 传输方式：none（无 ICE 处理）、relay（仅 TURN 服务）或 all（可接受 STUN 和 TURN）。默认值为 all。	属性	PC
RTCPeerConnection.updateIce() 使浏览器 ICE 代理重新启动或更新其本地候选项和远程候选项的集合，具体取决于指定的参数。	方法	PC
RTCPeerConnection.addIceCandidate() 向浏览器 ICE 代理提供远程候选项。	方法	PC
RTCPeerConnection.getConfiguration() 返回当前的 RTCConfiguration。	方法	PC
RTCPeerConnection.iceGatheringState 承载 ICE 代理当前在收集候选项地址方面的状态：new、gathering 和 complete。	属性	PC
RTCPeerConnection.iceConnectionState 承载 ICE 代理当前在连接两个对等端方面的状态：new、checking、connected、completed、failed、disconnected 和 closed。	属性	PC
RTCPeerConnection.onicecandidate 可由应用程序设置为一个函数 / 方法，每当有新的 ICE 候选项可发送给远程对等端时，都调用该函数 / 方法。这非常适用于“缓慢型 ICE”（Trickle ICE）。	属性	PC
RTCPeerConnection.oniceconnectionstatechange 可由应用程序设置为一个函数 / 方法，每当 RTCPeerConnection.RTCIceConnectionState 发生更改时，都调用该函数 / 方法。	属性	PC
RTCPeerConnectionIceEvent 当浏览器识别到 RTCIceCandidate 时返回 RTCIceCandidate。由 onicecandidate 处理。	事件	PC
RTCPeerConnectionIceEvent.candidate 浏览器识别的 RTCIceCandidate。	属性	PC

表 8.5 WebRTC 数据通道 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

数据通道 API 和说明	类型	参考
RTCPeerConnection.createDataChannel() 在 RTCPeerConnection 上创建新的数据通道。	方法	PC
RTCPeerConnection.ondatachannel 可由应用程序设置为一个函数 / 方法，每当创建 RTCDataChannel 时，都调用该函数 / 方法。	属性	PC
RTCDataChannelInit 包含用于创建数据通道的配置参数。	字典	PC
RTCDataChannel 表示 RTCPeerConnection 上可传输任意应用程序数据的通道。	接口	PC
RTCDataChannel.label 此数据通道的字符串标签，由应用程序在数据通道创建时设置。	属性	PC
RTCDataChannel.ordered 一个布尔值，由应用程序在此数据通道创建时设置，指示是否必须按数序传输消息。如果按顺序传输消息，则可能需要延迟传输某些消息。	属性	PC

（续）

数据通道 API 和说明	类型	参考
RTCDataChannel.maxRetransmitTime 对于不可靠的数据通道，此属性承载浏览器继续尝试重新发送数据所持续的毫秒数。此值由应用程序在数据通道首次创建时配置。请注意，如果同时设置此属性和 RTCDataChannel.maxRetransmits，将导致错误。	属性	PC
RTCDataChannel.maxRetransmits 对于不可靠的数据通道，此属性承载浏览器尝试重新传输数据的最大次数。此值由应用程序在数据通道首次创建时配置。请注意，如果同时设置此属性和 RTCDataChannel.maxRetransmitTime，将导致错误。	属性	PC
RTCDataChannel.protocol 规范尚未对此属性做出明确规定。请勿使用。	属性	PC
RTCDataChannel.negotiated 一个布尔值，指示是否自动协商此数据通道（即不进行显式 SDP 交换）。它反映了应用程序在数据通道创建时配置的值。	属性	PC
RTCDataChannel.id 创建数据通道时为通道设定的数值。它由浏览器自动确定，除非应用程序在通道创建时的配置中提供了此值。	属性	PC
RTCDataChannel.readyState RTCDataChannel 的状态。其值为下列选项之一：connecting、open、closing 和 closed。	属性	PC
RTCDataChannel.bufferedAmount 尚未传输且排队等待发送（由 RTCDataChannel.send() 发送）的字节数。	属性	PC
RTCDataChannel.onopen 可由应用程序设置为一个函数 / 方法，当数据通道准备好传输数据时，将调用该函数 / 方法。	属性	PC
RTCDataChannel.onerror 可由应用程序设置为一个函数 / 方法，每当数据通道的功能发生错误时，都调用该函数 / 方法。	属性	PC
RTCDataChannel.onclose 可由应用程序设置为一个函数 / 方法，当数据通道关闭时，将调用该函数 / 方法。	属性	PC
RTCDataChannel.close() 关闭数据通道。	方法	PC
RTCDataChannel.onmessage 可由应用程序设置为一个函数 / 方法，当在数据通道中收到消息（数据）时，将调用该函数 / 方法。	属性	PC
RTCDataChannel.binaryType 一个字符串，指示如何将二进制数据公开给应用程序。默认值为“blob”。	属性	PC
RTCDataChannel.send() 通过数据通道将参数发送给远程端。参数可采取多种格式，但最常用的两种格式是字符串和 blob。	方法	PC
RTCDataChannelEvent 当远程对等端创建了 RTCDataChannel 时返回 RTCDataChannel。由 ondatachannel 处理。	事件	PC
RTCDataChannelEvent.channel 由远程对等端创建的 RTCDataChannel。	属性	PC

表 8.6 WebRTC DTMF 处理 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

DTMF 处理 API 和说明	类型	参考
RTCPeerConnection.createDTMFSender() 对于指定为输入的 MediaStreamTrack，此方法会创建一个 RTCDTMFSender 对象，用于将 DTMF 提示音注入 RTCPeerConnection 的轨道表示形式。	方法	PC
RTCDTMFSender 用于将 DTMF 插入音频轨道的容器和控制对象。	接口	PC
RTCDTMFSender.canInsertDTMF 此布尔值指示关联的轨道是否能够插入 DTMF。此值通常为 true；如果轨道或对等连接存在问题，此值可能会变为 false。	属性	PC
RTCDTMFSender.insertDTMF() 将给定的 DTMF 字符串插入关联的轨道。请注意，此调用将替代 toneBuffer 中其他将要播放的所有提示音。	方法	PC
RTCDTMFSender.track 与此对象关联的 MediaStreamTrack。	属性	PC
RTCDTMFSender.ontonechange 可由应用程序设置为一个函数 / 方法，当在数据通道中收到消息（数据）时，将调用该函数 / 方法。	属性	PC
RTCDTMFSender.toneBuffer 包含将要播放的其他提示音。	属性	PC
RTCDTMFSender.duration 每个提示音应持续播放的毫秒数。默认值为 100 毫秒。	属性	PC
RTCDTMFSender.interToneGap 一个提示音结束与下一个提示音开始之间所间隔的毫秒数。默认值为 50 毫秒。	属性	PC
RTCDTMFSenderToneChangeEvent 返回刚刚开始发送的 DTMF 提示音。由 ontonechange 处理。	事件	PC
RTCDTMFSenderToneChangeEvent.tone 刚刚开始发送的 DTMF 提示音。	属性	PC

表 8.7 WebRTC 统计数据处理 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

统计数据处理 API 和说明	类型	参考
RTCPeerConnection.getStats() 对于指定为输入的 MediaStreamTrack，此方法将通过 RTCStatsCallback 处理程序收集并返回与轨道有关的传输统计数据。	方法	PC
RTCStatsCallback 可由应用程序设置为一个函数 / 方法，该函数 / 方法采用 RTCStatsReport 作为参数。由 RTCPeerConnection.getStats() 使用。	回调	PC
RTCStatsReport 包含一个或多个 RTCStats 对象，这些对象被用作输入传递给 getStats() 的轨道。由于一个轨道可使用一个或多个 SSRC 在对等连接中传输，因此该报告将为每个此类 SSRC 返回一个 RTCStats 对象。	接口	PC
RTCStats 一个特定对象的基类，该对象包含有关单个 RTP 对象类型的统计数据。此基类包含时间戳、ID 和类型。RTCRTPStreamStats 继承自此类。	字典	PC

（续）

统计数据处理 API 和说明	类型	参考
RTCRTPStreamStats 一个特定对象的父类，该对象包含有关单个 RTP 流的统计数据。此类包含流的另一端的 ID（因此便于检查其统计数据）以及此 RTP 流的 SSRC。此类继承自 RTCStats。RTCInboundRTPStreamStats 和 RTCOutboundRTPStreamStats 继承自此类。	字典	PC
RTCInboundRTPStreamStats 一个对象，其中包含单个入站 RTP 流的以下统计数据：自开始在对等连接上传输数据以来收到的数据包总数和字节总数。此对象继承自 RTCRTPStreamStats。	字典	PC
RTCOutboundRTPStreamStats 一个对象，其中包含单个出站 RTP 流的以下统计数据：自开始在对等连接上传输数据以来发送的数据包总数和字节总数。此对象继承自 RTCRTPStreamStats。	字典	PC

表 8.8 WebRTC 身份处理 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

身份处理 API 和说明	类型	参考
RTCConfiguration.requestIdentity 指示浏览器是否验证远程方的身份。可能的值包括 yes（必须请求身份）、no（不请求任何身份）和 ifconfigured（如果用户已在浏览器中配置身份，或如果已调用 setIdentityProvider()，则请求身份）。默认值为 ifconfigured。	属性	PC
RTCPeerConnection.setIdentityProvider() 允许应用程序为此对等连接指定身份提供程序、协议和声明的用户名（身份）。如果浏览器已配置这些信息，则可能不需要指定它们。	方法	PC
RTCPeerConnection.getIdentityAssertion() 启动身份提供程序进程（获取身份断言）。此方法为可选方法，但它会在提议或应答生成之前启动此进程，因而可能会加快应用程序的处理速度。	方法	PC
RTCPeerConnection.peerIdentity 此为远程对等端的 RTCIdentityAssertion。只有该对等端的身份已经过验证时，才设置此属性。	属性	PC
RTCPeerConnection.onidentityresult 可由应用程序设置为一个函数 / 方法，当验证身份的尝试成功或失败时，将调用该函数 / 方法。	属性	PC
RTCIdentityAssertion 包含对等端的经过验证的身份提供程序和身份（名称）。	字典	PC

表 8.9 WebRTC 流处理 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

流处理 API 和说明	类型	参考
MediaStream 表示 MediaStreamTrack 的集合，目前仅包含音频和视频。	接口	gUM
新的 MediaStream(MediaStream/ MediaStreamTrackSequence) 创建一个新的 MediaStream，其中包含其他 MediaStream 对象中的音频和视频轨道。输入参数为单个 MediaStream 或 MediaStreamTrackSequence，后者是 MediaStreamTrack 的数组。	构造函数	gUM
MediaStream.id 浏览器为此 MediaStream 生成的唯一标识符字符串；有关 MediaStream 的定义，请参见“媒体捕获和流”文档。“WebRTC 1.0”文档介绍了如何为远程端发起的流创建标签。	属性	gUM 和 PC

（续）

流处理 API 和说明	类型	参考
MediaStream.getAudioTracks() 返回一个数组，其中包含此 MediaStream 中的所有 AudioMediaStreamTrack。	方法	gUM
MediaStream.getVideoTracks() 返回一个数组，其中包含此 MediaStream 中的所有 VideoMediaStreamTrack。	方法	gUM
MediaStream.clone() 返回 MediaStream 的克隆，该克隆具有不同的 ID 以及 MediaStream 中所有轨道的克隆。	方法	gUM
MediaStream.ended 此属性由浏览器设置，当且仅当流已完成时，其值才为 true。	属性	gUM
MediaStream.onended 可由应用程序设置为一个函数 / 方法，当 MediaStream 完成时，将调用该函数 / 方法。	属性	gUM
MediaStreamEvent 返回由远程对等端添加或删除的 MediaStream。由 onaddstream 或 onremovestream 处理。	事件	PC
MediaStreamEvent.stream 由远程对等端添加或删除的 MediaStream。	属性	PC
URL 所有 Web 浏览器中预先存在的一个接口。	接口	gUM
URL.createObjectURL 对于指定为参数的 MediaStream，创建并返回一个“blob”URL。此 URL 将适合传递给 <audio> 元素（如果流包含音频）和 <video> 元素（如果流包含视频）。	方法	gUM
RTCPeerConnection.addStream() 将现有媒体流添加至 RTCPeerConnection 以发送至远程对等端。	方法	PC
RTCPeerConnection.removeStream() 从 RTCPeerConnection 中删除一个 RTCPeerConnection 流，这最终将导致不再发送流。	方法	PC
RTCPeerConnection.getLocalStreams() 返回一个数组，其中包含由本地端发起的所有 MediaStream 值。	方法	PC
RTCPeerConnection.getRemoteStreams() 返回一个数组，其中包含由远程端发起的所有 MediaStream 值。	方法	PC
RTCPeerConnection.getStreamById() 返回对等连接中具有指定 ID 的 MediaStream；如果该对象不存在，则返回 null。	方法	PC
RTCPeerConnection.onaddstream 可由应用程序设置为一个函数 / 方法，每当添加远程流时，都调用该函数 / 方法。	属性	PC
RTCPeerConnection.onremovestream 可由应用程序设置为一个函数 / 方法，每当删除远程流时，都调用该函数 / 方法。	属性	PC
NavigatorUserMedia 所有 Web 浏览器中预先存在的一个接口。	接口	gUM
NavigatorUserMedia.getUserMedia() 返回一个 MediaStream，其中包含满足指定为输入的约束（详见 MediaStreamConstraints）的一个或多个媒体轨道。	方法	gUM
NavigatorUserMediaSuccessCallback 可由应用程序设置为一个函数 / 方法，该函数 / 方法接受 MediaStream 作为参数。由 NavigatorUserMedia.getUserMedia() 使用。	回调	gUM

（续）

流处理 API 和说明	类型	参考
MediaError 表示调用 NavigatorUserMedia.getUserMedia() 时返回的错误。	接口	gUM
MediaError.name 调用 NavigatorUserMedia.getUserMedia() 时发生的错误。必须为“PERMISSION_DENIED”或“CONSTRAINT_NOT_SATISFIED”，前者指示用户不允许页面使用本地设备，后者指示无法满足强制性约束。	属性	gUM
MediaError.message 可供用户阅读的、关于所发生的错误的说明。	属性	gUM
MediaError.constraintName 如果错误名称为“CONSTRAINT_NOT_SATISFIED”，此属性的值将为导致错误的约束。	属性	gUM
NavigatorUserMediaErrorCallback 可由应用程序设置为一个函数 / 方法，该函数 / 方法接受 NavigatorUserMediaError 作为参数。由 NavigatorUserMedia.getUserMedia() 使用。	回调	gUM

表 8.10 WebRTC 轨道处理 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

轨道处理 API 和说明	类型	参考
MediaStreamTrack 表示媒体源的单个轨道。请注意，每个轨道可包含多个通道，例如编码为单个轨道的 6 声道环绕声源。此外，每个轨道只能包含一种媒体，而不论它有多少个通道。	接口	gUM
MediaStreamTrack.kind 值为 audio 或 video。	属性	gUM
MediaStreamTrack.id 由浏览器生成的全局唯一标识符。	属性	gUM
MediaStreamTrack.label 由浏览器为此 MediaStreamTrack 生成的标签字符串，例如“Built-in microphone”（内置麦克风）。浏览器可选择提供空字符串之外的任何字符串作为标签。	属性	gUM
MediaStreamTrack.enabled 可由应用程序设置的布尔值，用于禁用或重新启用轨道输出。	属性	gUM
MediaStreamTrack.muted 一个布尔值，指示是否将轨道设置为静音。	属性	gUM
MediaStreamTrackState.live MediaStreamTrack.readyState 可能具有的值之一，指示轨道处于活动状态，即能够生成输出。请注意，即使在活动状态下，MediaStreamTrack 也可被禁用（详见 MediaStreamTrack.enabled）或设置为静音（详见 MediaStreamTrack.muted），因此不会生成输出。	板举	gUM
MediaStreamTrackState.new MediaStreamTrack.readyState 可能具有的值之一，指示轨道尚未连接至源。	板举	gUM
MediaStreamTrack.onmute 可由应用程序设置为一个函数 / 方法，每当 MediaStreamTrack 设置为静音时，都调用该函数 / 方法。	属性	gUM
MediaStreamTrackState.ended MediaStreamTrack.readyState 可能具有的值之一，指示轨道已结束，即不再能够且永远不会生成输出。	板举	gUM

（续）

轨道处理 API 和说明	类型	参考
MediaStreamTrack.onunmute 可由应用程序设置为一个函数 / 方法，每当 MediaStreamTrack 被取消静音时，都调用该函数 / 方法。	属性	gUM
MediaStreamTrack.onstarted 可由应用程序设置为一个函数 / 方法，当 MediaStreamTrack 变为活动状态时，将调用该函数 / 方法。	属性	gUM
MediaStreamTrack.onended 可由应用程序设置为一个函数 / 方法，当 MediaStreamTrack 完成时，将调用该函数 / 方法。	属性	gUM
MediaStreamTrack.readonly 一个布尔值，指示轨道是否不可约束。例如，在对等连接上收到的文件或某些轨道。	属性	gUM
MediaStreamTrack.remote 一个布尔值，指示轨道是否由对等连接的远程端发起。	属性	gUM
MediaStreamTrack.readyState 指示轨道的状态：new、live 或 ended。有关定义，请参见“媒体捕获和流”文档。“ WebRTC 1.0”文档介绍了为何必须设置由远程端发起的轨道属性。	属性	gUM 和 PC
MediaStreamTrack.clone() 创建拥有自己 ID 的轨道副本。	方法	gUM
MediaStreamTrack.stop() 结束轨道。如果没有其他轨道引用媒体源，则停止相关源并关闭所有指示设备正在使用的通知。	方法	gUM
MediaStreamTrackEvent 在添加或删除轨道时返回一个 MediaStreamTrack。由 onaddtrack 或 onremovetrack 处理。	事件	gUM
AudioMediaStreamTrack MediaStreamTrack 的子类，只能承载 sourceType 为“音频”的媒体。	接口	gUM
AudioMediaStreamTrack.getSourceIds() 一个静态方法，可返回所有可用音频源的 ID。	方法	gUM
VideoMediaStreamTrack MediaStreamTrack 的子类，只能承载 sourceType 为“视频”的媒体。	接口	gUM
VideoMediaStreamTrack.getSourceIds() 一个静态方法，可返回所有可用视频源的 ID。	方法	gUM
MediaStream.getTrackById() 返回此 MediaStream 中具有指定 ID 的 MediaStreamTrack；如果不存在此类轨道，则返回 null。	方法	gUM
MediaStream.addTrack() 如果指定轨道尚不存在，则将其作为参数添加至 MediaStream。	方法	gUM
MediaStream.removeTrack() 从 MediaStream 中删除指定为参数的轨道。	方法	gUM
MediaStream.onaddtrack 可由应用程序设置为一个函数 / 方法，每当向此 MediaStream 中添加轨道时，都调用该函数 / 方法。	属性	gUM
MediaStream.onremovetrack 可由应用程序设置为一个函数 / 方法，每当从此 MediaStream 中删除轨道时，都调用该函数 / 方法。	属性	gUM

表 8.11 WebRTC 约束和功能 API（PC= 对等连接 [1]；gUM=getUserMedia [2]）

约束与功能 API 和说明	类型	参考
NavigatorUserMedia.getMediaDevices() 返回可供应用程序在创建正常运行的音频 / 视频输入选择器对话框时使用的源标识信息。	方法	gUM
MediaDeviceInfoCallback 可由应用程序设置为一个函数 / 方法，该函数 / 方法接受 MediaDeviceInfo 作为参数。由 NavigatorUserMedia.getMediaDevices() 使用。	回调	gUM
Constrainable 添加用于操作约束、功能和设置的方法和属性。由 MediaStreamTrack 实现。	接口	gUM
Constrainable.getConstraints() 返回当前应用于此对象的约束（如果有）。	方法	gUM
Constrainable.getSettings() 返回所有可约束属性的当前值。	方法	gUM
Constrainable.getCapabilities() 返回此对象的各种可能的约束所支持的值。	方法	gUM
Constrainable.applyConstraints() 尝试将对象的当前约束替换为以参数形式指定的约束。	方法	gUM
Constrainable.onoverconstrained 可由应用程序设置为一个函数 / 方法，每当此对象约束过度时，都调用该函数 / 方法。	属性	gUM
MediaStreamTrack.getConstraints() 请参见 Constrainable.getConstraints()。	方法	gUM
MediaStreamTrack.getSettingsow () 请参见 Constrainable.getSettings()。	方法	gUM
MediaStreamTrack.getCapabilities() 请参见 Constrainable.getCapabilities()。	方法	gUM
MediaStreamTrack.applyConstraints() 请参见 Constrainable.applyConstraints()。	方法	gUM
MediaStreamTrack.onoverconstrained 请参见 Constrainable.onoverconstrained。	属性	gUM
MediaErrorEvent 返回一个 MediaError 对象，其中包含错误名称和消息；如果错误名称为“ CONSTRAINT_NOT_SATISFIED”，则还会一并返回未满足的强制性约束。	事件	gUM
MediaDeviceInfo 包含有关源或接收器的有助于消除歧义的简单信息。	字典	gUM
MediaDeviceInfo.deviceId 此源或接收器的唯一 ID。请参见 MediaStreamTrack.id。	属性	gUM
MediaDeviceInfo.kind 值为“audioinput”、“audiooutput”或“videoinput”。	属性	gUM
MediaDeviceInfo.label 浏览器为源或接收器提供的标签。请参见 MediaStreamTrack.label。	属性	gUM

8.2 WEBRTC 建议

目前，尚没有 WEBRTC 规范达到“建议”状态。

8.3 WEBRTC 草案

所有 WEBRTC 文档当前都处于“工作草案”阶段。接下来的几节将介绍这些文档。请注意，最好结合规范文档本身来阅读以下说明，因为这些说明并没有重复规范中的全部内容。

8.3.1 WebRTC 1.0：浏览器之间的实时通信

此文档 [1] 是 WebRTC 研究工作的主要文档，私下通常称为“对等连接”草案。其中定义了 RTCPeerConnection 接口，并对“媒体捕获和流”（getusermedia）规范中定义的 MediaStream 接口进行了扩展。RTCPeerConnection 接口中定义了许多功能。为便于读者阅读，该规范将相关功能划分到不同的章节中；读者不要因此而天真地认为，RTCPeerConnection 定义中只包含很少的方法和属性。在本节中，我们将介绍这些经过分组的功能集合。它们包括：核心 RTCPeerConnection 本身、数据 API、DTMF API、统计数据 API 和身份 API。

8.3.1.1 RTCPeerConnection 接口

RTCPeerConnection 接口是 WebRTC 技术的主要 API。此 API 的功能是在两个浏览器之间建立媒体连接路径。虽然此 API 紧密绑定到 JavaScript 会话建立协议（JavaScript Session Establishment Protocol，JSEP）（详见 11.3.8 节）以进行媒体协商，但 JSEP 的大多数细节都由浏览器处理。

RTCPeerConnection 对象的构造函数带有一个 RTCConfiguration 对象，该配置对象具有多个实用的属性。第一个属性是 IceServers，可设置为服务器地址的数组，用于帮助通过 NAT 和防火墙（STUN 和 TURN 服务器，详见 10.2.5 节和 10.2.6 节）建立会话。IceTransports 属性可设置为三个值之一：all——指示检索到的任何候选项均可用于 ICE；relay——指示只有中继候选项可用于 ICE；none——指示任何候选项都不能用于 ICE。requestIdentity 属性由身份功能（详见 8.3.1.5 节）用于确定本地浏览器在哪些情况下需要获取对等端用户的身份，其值可以是 yes、no 或 ifconfigured。稍后可使用 RTCPeerConnection 的 getConfiguration() 方法检索此配置对象。

当然，RTCPeerConnection 对象可以有关联的媒体流。媒体流的添加和删除分别使用 addStream() 和 removeStream() 方法实现。请注意，媒体流并非这些方法所创建；这些方法只是在要发送给远程对等端的流集合中添加和删除现有本地流。getLocalStreams() 和 getRemoteStreams() 方法可用于提取相应的数组，以分别跟踪整套本地流和远程流。此对等连接 API 有一个不十分明显但很重要的方面，即 SDP 会话协商由 Web 应用程序负责管

理。换言之，addStream() 和 removeStream() 不会导致媒体开始流动或停止流动。它们可更改本地 RTCPeerConnection 对象的状态，但需要进行显式的对话协商才能与远程端协调媒体更改。为触发应用程序代码执行协商，addStream() 和 removeStream() 方法会引发 negotiationneeded 事件。当应用程序捕获此事件（或设置 onnegotiationneeded 回调）时，应用程序必须通过以下方式协商媒体：

1）调用 createOffer()——用户代理将检查对等连接的内部状态，并生成相应的 RTCSessionDescription 对象（提议）。

2）对 RTCSessionDescription 对象调用 setLocalDescription()。

3）将生成的 SDP 会话描述发送给远程对等端。请注意，此规范并未定义或规定在与远程对等端之间发送 SDP 会话描述时使用何种机制。此规范将这种未定义的通道称为“信令通道”，详见第 4 章。

当然，如果远程对等端是发送提议的一方，应用程序将需要改为调用 createAnswer() 并发回生成的 RTCSessionDescription 对象。当在信令通道中收到远程端的提议（或应答）时，应用程序必须调用 setRemoteDescription() 才能完成提议 / 应答协商。无论是本地端还是远程端，只要浏览器中成功应用了最终应答，实际媒体状态就会发生改变。

此时，读者显然会问：为何信令通道的设置和分析以及全部协商工作都交由应用程序执行？这种设计的主要目的是提高灵活性。从信令角度而言，许多实现浏览器到浏览器通信的应用程序让两个浏览器都使用来自同一 Web 服务器的完全相同的源代码，从而能够以符合逻辑的方式通过该服务器进行信令传输。有些应用程序则可能需要通过网关传输信令。从 SDP 会话协商角度而言，将这一切交由应用程序处理有两个最大的优势：一是可以实现“缓慢型”ICE，其中 ICE 处理甚至可以在生成全部 ICE 候选项之前进行（详见 11.5.1 节）；二是能够根据需要修改 SDP。由于 SIP/SDP 互操作尚未完全实现，因此经常需要调整 SDP，尤其是在浏览器与非浏览器端点进行通信时。为支持最常见的信令和协商使用情形，可能会开发出相应的库。

对等连接的一个复杂之处在于，它有两个进程：一是 ICE 进程，二是提议 / 应答媒体协商进程；这两个进程都有各自的状态机。提议 / 应答状态机由 JavaScript 进程控制，而 ICE 状态机则由浏览器控制。SDP 中反映的会话描述带有提议或应答的媒体，以及用于 ICE“打洞”的候选项。但是，ICE 进程并不依赖于提议 / 应答进程，因此进展缓慢的 ICE 进程有时会在媒体协商完成（即不需要进一步交换 SDP 即可就媒体达成一致）后仍继续检查额外的候选项。在实际中，虽然底层 ICE 状态机可能不止一个（具体数量取决于指定的媒体传输方式），但 WebRTC API 只公开一个合并的状态机。根据规范，ICE 处理的进度实际上分为两部分——候选项收集以及检查和连接；这是因为它们可在一定程度上并行执行。iceGatheringState 的值只包括 new、gathering 和 complete。iceConnectionState 的值包括 new、checking、connected、completed、failed、disconnected 和 closed。同样，这些状态的依次变迁大都与 JavaScript 代码无关。但是，如果能够检查这些状态变量，将有助于构造可

靠的应用程序。signalingState 变量跟踪提议 / 应答交换状态，其值包括 stable、have-local-offer、have-remote-offer、havelocal- pranswer、have-remote-pranswer 和 closed。此规范中提供了 signalingState 和 iceConnectionState 的状态过渡图表。图 11.9 中的 JSEP 信令状态机图表非常类似于 signalingState 的过渡图表。

由于应用程序可能需要更改其希望使用的 ICE 服务器集合或允许使用的传输方式，因此现在还有一个 iceUpdate() 方法，此方法与 RTCPeerConnection 构造函数接受同一 RTCConfiguration 对象。

8.3.1.2 DataChannel 接口

RTCDataChannel 接口是 WebRTC 规范中定义的一项关键功能，此 API 用于在对等连接中建立双向数据通道。为创建 RTCDataChannel，此规范提供了巧妙命名的 RTCPeerConnection.createDataChannel() 方法。除了可由应用程序设置的标签外，此方法还带有一个可选的 dataChannelDict 配置对象。通过该对象的各种属性可控制两个主要特征：一是通道的可靠性；二是顺序是否重要。在理想状态下，可靠的顺序传输不会产生任何成本。但在实践中，完美的可靠性需要不限制重新传输的次数，而顺序传输则需要无限的缓冲来收集无序的消息。maxRetransmitTime 和 maxRetransmits 这两个属性用于限制可靠性。前者允许应用程序指定浏览器继续重新传输所持续的最大时间长度（以毫秒为单位）。后者允许应用程序指定尝试重新传输的最大总次数。在这两个属性中，只能指定一个（否则可能导致错误）。对于布尔型 ordered 属性，你可能已经猜到，它允许应用程序指定数据通道是否按顺序传输消息。此配置对象还允许应用程序设置 ID（或由浏览器自动选择 ID）并指定新数据通道的协商将由浏览器（默认协商主体）而非应用程序处理。

创建新 RTCDataChannel 的进度由其 readyState 属性跟踪，该属性的值包括 connecting（正在建立数据通道）、open（数据通道已经就绪，可随时使用）、closing（正在关闭数据通道）和 closed（数据通道已经关闭或者从未建立）。显然，RTCDataChannel 有一个 close() 方法，以及可针对 onopen、onclose 和 onerror 设置的处理程序。

数据通道建立之后，将展现出类似于 WebSocket 的行为：通过 send() 方法发送数据，并能够对传入的消息设置 onmessage 处理程序。目前，关于数据通道的部分细节尚未最终确定，但其总体结构已十分稳定。

8.3.1.3 DTMF API

在连接到当今的电话网络时，音频通信通道需要能够传达双音多频（Dual Tone Multi-Frequency，DTMF）提示音。具体而言，向 SIP 基础架构中发送音频的 JavaScript 应用程序将需要一种方式来生成 DTMF 提示音，因为音频可能来自普通麦克风或其他没有 DTMF 小键盘的设备。但是，只有在为对等连接协商媒体期间，才会安排同时发送 RFC 4733 DTMF 数据包的功能。因此，此规范在 RTCPeerConnection 上提供了一个接口，用于将 DTMF 功能与该 RTCPeerConnection 的指定 MediaStreamTrack 相关联。此部分向 RTCPeerConnection

接口中添加了一个以 MediaStreamTrack 为参数的方法 createDTMFSender()。随后，便可通过新 RTCDTMFSender 对象的 insertDTMF() 方法在对等连接中发送该 MediaStreamTrack 中的 DTMF 提示音。请注意，DTMF 只能通过 RTCDTMFSender 插入，而不能通过原始 MediaStreamTrack 本身的任何方法直接插入。

8.3.1.4 统计数据 API

此 WebRTC 规范定义了一个 API，用于从要通过对等连接传输的轨道中收集统计数据。这些数据可能十分重要，例如确定当前有多少数据包被送达。此 API 的工作方式如下：getStats() 将 MediaStreamTrack 作为输入，并带有一个将在统计数据可用时执行的回调。该回调将被提供一个包含 RTCStats 对象的 RTCStatsReport。统计数据将按类型划分到这些对象中，但所有这些类型都包含 ID、类型和时间戳属性。当前只定义了两个类型：inbound-rtp 和 outbound-rtp。这两个类型都是 RTCRTPStreamStats 子类的实例，该子类额外提供 remoteId 和 ssrc 属性。outbound-rtp 对象类型由子类 RTCOutboundRTPStreamStats 表示，该子类提供 packetsSent 和 bytesSent 属性。inbound-rtp 对象类型理所当然由子类 RTCInboundRTPStreamStats 表示，该子类提供类似的 packetsReceived 和 packetsSent 属性。通过结合使用这些属性，可以跟踪当前发送和接收了多少关于轨道的数据。请注意，remoteId 属性可直接用作 RTCStatsReport 中的键来提取当前轨道远程端的报告。对于报告中的对象，无论其属于何种类型，都可以使用 for 循环来访问，从而轻松处理它们。

8.3.1.5 身份 API

此 WebRTC 规范中的身份 API 与其他 API 略有不同，它假定浏览器已经使用声明的身份进行配置，并配置有身份提供程序，因此如果在底层使用了 Web 身份提供程序，应该不需要 Web 应用程序执行任何操作。从理论上讲，应用程序只需设置每次在验证身份时调用的 onidentityresult 处理程序即可，因为一端的浏览器可自动为配置的身份生成签名，而另一端的浏览器可自动拉出身份提供程序的验证页面来验证签名。但在实践中，描述底层协议要求的 IETF 草案（详见 13.4 节）建议，签名本身需要使用信令通道传输。由于信令通道只由应用程序 JavaScript 代码直接使用，因此至少需要在此步骤中引入应用程序代码。总之，此规范在一些方面仍没有规定清楚。无论如何启动签名过程及如何表达和验证签名，身份 API 无疑都为应用程序代码提供了一种方式来重写浏览器的已配置身份和身份提供程序，其途径便是对等连接上的 setIdentityProvider() 方法。此 API 还为需要在提议 / 应答交换之前启动身份检查过程（当该交换自动进行时，就需要这样处理）的应用程序提供了 getIdentityAssertion() 方法。当用户登录需要较长时间，并因此导致身份验证时间过长时，该方法可能有助于提高用户界面的响应速度。此身份机制目前还非常不稳定，而且尚没有实例进行了可靠的实现，因此暂时最好不要依赖此机制。

8.3.1.6 MediaStream 接口扩展项

此 WebRTC 规范还为“媒体捕获和流”规范（详见 8.3.2 节）中定义的 MediaStream

接口定义了扩展项。首先，每个MediaStream都有一个ID属性用于区别自身和通过RTCPeerConnection API发送的其他MediaStream。第二，当远程端在现有MediaStream中添加或删除MediaStreamTrack时，将在本地端生成addtrack和removetrack事件。同样，本地端还会复制远程端的静音和取消静音事件。

8.3.2 媒体捕获和流

此文档[2]由媒体捕获任务组负责，该任务组由WEBRTC工作组以及设备API和策略（Device APIs and Policy，DAP）工作组的成员构成。此文档之前称为getusermedia草案，其中定义了getUserMedia()方法，此API用于从摄像头或麦克风等设备中获取本地媒体流。

8.3.2.1 getUserMedia()方法

此API旨在供所有Web应用程序开发人员用作访问本地设备媒体的主要API。因此，它要求浏览器在访问设备之前获取用户许可。但是，获取此许可的机制、许可的具体指定方式、许可的持续时间以及有关许可的所有其他细节都由用户代理负责处理。建议用户代理在本地设备进行录制时提供显著的指示。对于设置成功和失败的情况，分别有两个不同的回调：successCallback和errorCallback。后者会返回MediaError，该对象包含名称（错误名称）、消息（可供用户阅读的错误字符串）和constraintName（未满足的强制性约束（如果有）的名称）属性。关于具体存在哪些错误，目前仍有一些混乱，但目前可以确定，将有一个错误名为ConstraintNotSatisfiedError。

此规范最近添加了getMediaDevices()方法，它将取代之前的getSourceInfos()方法。通过这一新方法，应用程序能够获取可供自己使用的外部源和接收器的初始列表。请注意，同一个实际物理设备可能会被浏览器呈现为多个不同的源。getMediaDevices()带有一个回调，当调用该回调时，将提供一个MediaDeviceInfo对象列表。每个对象都包含以下属性：deviceId（与sourceId相同）、kind（指示音频/视频和输入/输出）、label和groupId。自从此规范编制伊始，研究人员就非常担心应用程序获取设备列表的功能会影响隐私和用户指纹识别，并在此方面进行了大量讨论。MediaDeviceInfo对象是目前一种折中的方法。它提供的标签非常适合在应用程序向用户显示的选择器中使用，但没有足够的细节来用于识别指纹。groupId十分有趣，与getSourceInfos更名为getMediaDevices这一情况直接相关。虽然“媒体捕获”规范定义了如何选择和控制输入媒体源，但也有不少人请求提供输出设备选择和控制机制。由于其他W3C工作组也表达了同样的需求，我们目前最多只能决定，允许获取输入源和输出接收器的列表，并提供一个groupId将彼此相连的输入和输出关联起来。例如，耳机/麦克风组合可能被表示为一个输入源和一个输出接收器，各自具有不同的deviceId，但具有相同的groupId。

8.3.2.2 设置、功能和约束

关于getUserMedia()的constraints参数，有必要对其格式和处理进行单独说明。此

参数包含一个 JavaScript 对象，且每个媒体类型（当前只有音频和视频）都有一个属性。每个类型的值可以是一个布尔值（稍后说明）或一个对象。该对象具有两个可选属性：mandatory 和 optional。第一个属性包含一个 ConstraintSet，此集合是必须满足的约束键和值；如果不满足它们，将会返回错误。第二个属性包含一个按优先级排序的 ConstraintSet 对象列表。未能满足一个或多个可选约束并不会导致错误，但这里有一项要求，即当约束之间发生冲突时，应优先满足序列中排在前面的约束。如果赋予媒体类型的值不是对象，而是布尔值 true，例如 video:true，则必须获取视频轨道，否则将返回错误。在此规范最终敲定之前，潜在约束的列表可能会扩展，也可能会更改，但就目前而言，这是最新的列表。对于音频和视频，sourceId 使用标识符来标识源，同一来源的标识符将在各会话之间保持不变，但不同于可供浏览器使用的其他来源。视频约束包括 width、height、frameRate、aspectRatio 和 facingMode。此规范对这些约束做了详细规定，不过还是有必要说明一下 facingMode。关于应设置哪些选项来描述摄像头的朝向，研究人员进行了大量讨论。虽然他们考虑了“向前”、“向后”、“向左”和“向右”，但只有最后两个没有被修改。对于“向前”和“向后”，我们现在使用“环境”（背对用户）和“用户”（面向用户）来代替它们。当前音频约束包括 volume、sampleRate、sampleSize 和布尔型的 echoCancelation。

与之前提议的使用可选提示的较简单 API 相比，这种通过约束来选择和控制轨道与其他属性的功能有两个优点：一是应用程序能够提前指示不能接受哪些特定的轨道；二是能够指示某些约束是否比其他约束更为重要。出于这些原因，并为了以后能够进行扩展，getUserMedia() 接受上一段中介绍的 constraints 参数。

每个约束都有一项关联的功能和一项关联的设置，二者在“媒体捕获和流”规范中都有相当详尽的说明。功能是约束的有效作用范围。设置是与该约束相对应的源的当前设置 / 值。虽然功能、设置和约束都通过 MediaStreamTrack 的方法和属性访问，前两者却恰好是馈送轨道的媒体源的属性。换言之，共享同一源的两个轨道将具有相同的状态值。下面我们以视频宽度为例来进行说明。

摘自某约束说明：

```
{
  mandatory: {
    width: { max: 1280 }
  },
  optional: [
    { width: 1000 },
    { width: { min: 640 }}
  ]
}
```

在此示例中，其结构要求视频轨道的宽度尽量正好为 1000 像素（即最小值和最大值均为 1000）。如果不满足此要求，其宽度至少应大于或等于 640。最重要的是，宽度不得超过 1280 像素。如果源提供的宽度超过了 1280 像素，则约束请求一定失败。

作为设置值：

```
mytrack.getSettings().width 将返回轨道的视频宽度。
```

作为功能：

mytrack.getCapabilities().width 可能会返回如下结构：

```
{ max:1920, min:640, supported: true }
```

此规范提供了一个重要的跟踪方法 applyConstraints()，此方法允许动态更改约束。由于轨道对源施加约束，而这些约束又可能发生动态更改，因此轨道可能变为约束过度。当多个轨道与同一个源相关联时，就更可能发生这种情况，因为每个轨道都有机会对源施加约束，因而可能会导致冲突。此规范对这一主题进行了相当详尽的讨论。这里需要注意的一个主要问题是，轨道随时可能变为约束过度，届时将为该轨道生成 overconstrained 事件并将其设置为静音。此事件无疑会被捕获并处理。

与最初提案相比，约束、功能和设置在指定方式上发生了重大变化，而约束的核心工作机制却几乎没变。阅读此规范时，读者可能会感到混乱，不清楚每项内容具体在何处定义。约束机制在此规范末尾单独定义的 Constrainable 接口中统一介绍。现在，轨道支持所有这些方法，由此可以得到的唯一提示是，MediaStreamTrack 规范实现了 Constrainable 接口。之所以单独定义，是为了让其他与 WebRTC 相关的规范也能够利用约束机制。

8.3.2.3 MediaStream 接口

此文档还定义了 MediaStream 接口和 MediaStreamTrack 接口，前一 API 用于创建表示媒体数据流的对象。有关这两个接口的详情，请参见 3.1.1 节和 3.1.2 节。

8.3.2.4 MediaStreamTrack 接口

此 API 的基本单位是 MediaStreamTrack。该轨道代表单个设备或录制内容可返回的单一类型的媒体。单个立体声源或 6 声道环绕声音频信号均可视为一个轨道，尽管二者都由多个音频声道构成。请注意，此规范虽然粗略地规定了各声道“彼此具有公认的关系”，但并未定义在声道级别访问或操作媒体的方式。实际上，根据 WebRTC 文档中的定义，轨道的内容“将一起进行编码，以便作为某种有效负载类型（例如 RTP）进行传输”。换言之，在使用对等连接进行传输时，轨道的各个声道将被视为一个单元，即使其在本地处于启用 / 禁用或静音状态，情况也不例外。

此前，无法直接创建 MediaStreamTrack；此对象没有相应的构造函数。现在，此规范定义了 AudioMediaStreamTrack 和 VideoMediaStreamTrack 子类，二者都有构造函数，不过尚没有浏览器实现这一点。不久，getUserMedia() 方法可能会接受一个或多个新创建的轨道和 / 或流作为输入参数，其中这些轨道和 / 或流将被附加到要获取的媒体中。关于创建轨道（当前已实现）的其他方式，将在下面介绍媒体流的章节中介绍。MediaStreamTrack 有一个非常有趣的方面，即它实质上只是底层中由浏览器托管的媒体源的句柄。因此，

MediaStreamTrack 对象可能与其底层源相分离。此外，不同的 MediaStreamTrack 对象可代表同一媒体源，详见以下介绍 MediaStream 对象的章节。有两种方式可用于暂停轨道的媒体：静音和禁用。将轨道静音 / 取消静音这一操作由用户和 / 或浏览器执行，而静音则表示轨道的底层媒体源暂时无法提供媒体。例如，如果用户通过单击浏览器 Chrome 中的静音按钮或者切换手机侧面的开关来暂停媒体源的使用权限，就会出现这种情况。一般而言，应用程序无法控制何时将轨道静音。但是，它可以检查轨道的 muted 属性值。静音后，音频轨道将不再发声，视频轨道将显示黑屏。通过将轨道对象的 enabled 属性设置为 false，可单独逐一禁用每个轨道。这两个属性均独立于轨道的 readyState 属性；readyState 属性表示轨道的状态——new、live 或 ended。如果轨道的状态为 new，则表示其尚未连接至媒体；如果轨道的状态为 ended，则表示其源当前没有且永远无法再提供更多数据。例如，如果拔掉正在使用的摄像头的电源，就会出现这种情况。如果轨道的状态为 live，则表示其可以生成媒体。由于这些属性彼此独立，因而可以同时存在，例如，轨道可同时具有 live、unmuted 和 disabled 属性。

8.3.3 MediaStream 捕获情形

此文档 [3] 定义了有关媒体捕获和媒体流的要求与使用情形。其中的要求分为以下四个类别：权限、本地媒体、远程媒体和媒体捕获。目前，这些要求仍在审核之中，不久还可能加入 IETF RTCWEB 使用情形和要求草案文档（详见 11.3.2 节）中的要求。

8.4 相关工作

关于 W3C，研究人员正在积极讨论是否将其某些方面纳入 WebRTC 规范之一，另有一些方面则尚未纳入规范的编辑草案或公开草案。接下来的几节将简要介绍这些内容。有关 W3C 标准流程的详情，请参见附录 A。

8.4.1 MediaStream 录制 API 规范

WebRTC API 提供了设备轨道的句柄，但不提供对内容（数据）本身的访问。当然，可以通过对等连接将轨道的内容发送至可录制内容的实体，但有些正当的使用情形则要求直接访问数据。交互式语音响应（Interactive Voice Response，IVR）系统就是这样一种使用情形，这种系统可接听电话并播放语音菜单。许多此类 IVR 系统允许呼叫方使用语音播放菜单选项。录制 API 将允许 JavaScript 代码访问内容，以便进行保存、实施附加操作，或发布给自动语音识别系统。此文档的早期工作草案现已发布 [4]。

8.4.2 图像捕获 API

在讨论应在“媒体捕获和流”规范中定义哪些约束时，人们发现，应用程序开发人员

可能需要对视频 MediaStreams 进行多种处理和控制。有人提出了控制摄像头和通过摄像头捕获静态图像的方案 [5]，此方案今后可能会成为正式的工作草案。

8.4.3 future

最近，有人请求将所有使用回调的 API 方法都改为 future。简单而言，在计算机科学中，future 是一个变量，它暂时尚未赋值，但在未来的某个非指定时点将获得一个值。当前为 HTML DOM [6] 定义的 future 具有无阻断语义，根据这种语义，可定义处理程序，以便在 future 获取值时执行相应的操作。例如，不应调用

```
createOffer(gotSDP(), didntGetSDP())
```

而应调用

```
createOffer().done(gotSDP(), didntGetSDP())
```

凭借这种细微的语义差异以及其他一些功能，将能够以一种便利的方式检查异步结果组合。例如，如果将 getUserMedia() 和 createSignalingChannel() 都定义为 future，

```
Future.every(getUserMedia({video:true, audio:true}),
             createSignalingChannel()
            ).done(executeMe())
```

将等待二者都完成后才调用 executeMe()。虽然这一编程范例已在多种语言中应用多年，但它对 HTML 标准而言仍是一种新事物。鉴于这一原因，再加上工作组正在加紧敲定 WebRTC 规范的技术内容，因此这种相对较新的功能可能不会被纳入 WebRTC 标准的第一个版本。请注意，DOM Future 尚没有正式的 W3C 工作草案，因此其语法、内容乃至名称都可能发生改变。例如，人们正在考虑采用另一个名称 Promise（而非 future）。由于 ECMAScript 版本 6 本身刚刚增加了 Promise，因此这一名称改动迟早会发生。

8.4.4 媒体隐私

“媒体捕获和流”规范中定义的一项附加功能是 peerIdentity 属性，此属性可用于 getUserMedia() 约束结构。设置此属性后，它可使所获取的 MediaStream 及其轨道与所有应用程序隔离。具体而言，这意味着，只有浏览器对应用程序访问提供了与对跨源内容相同的限制时，才能在音频和视频元素中呈现 / 显示轨道（详见 13.2.3 节）。MediaStream 及其轨道也可以通过对等连接发送，但它们必须遵守正在制定的 WebRTC 规范中的额外要求。没错，规范的制定工作还没有结束。

8.4.5 MediaStream 的非活动状态

目前已经定义 MediaStream 在被附加到 <video> 等 HTML 元素时的行为。具体而言，即使 MediaStream 的现有轨道已经结束，也可以向其中添加新的轨道，因此需要对这种情

况下的行为进行规定。如果 MediaStream 包含未结束的轨道，则称其状态为活动；如果它的所有轨道都已结束，则称其状态为非活动。添加新的 live 轨道将使其再次变为活动状态。对于 <video> 元素，将出现如下结果：当附加到 <video> 元素的 MediaStream 变为非活动时，该元素将终止，且其 ended 属性设置为 true。此时，即使向 MediaStream 中添加了新的 live 轨道，该元素也不会重新开始播放媒体，除非应用程序明确重新开始播放（例如调用 play()）或 autoplay 属性被设置为 true。

8.5 参考资料

[1] 公开工作草案：http://www.w3.org/TR/webrtc ；编辑草案：http://dev.w3.org/2011/webrtc/editor/webrtc.html。

[2] 公开工作草案：http://www.w3.org/TR/mediacapture-streams ；编辑草案：http://dev.w3.org/2011/webrtc/editor/getusermedia.html。

[3] 公开工作草案：http://www.w3.org/TR/capture-scenarios ；编辑草案：http://dvcs.w3.org/hg/dap/raw-file/tip/media-streamcapture/scenarios.html。

[4] http://www.w3.org/TR/mediastream-recording。

[5] http://gmandyam.github.io/image-capture。

[6] http://dom.spec.whatwg.org/#futures。

第 9 章 Chapter 9

NAT 和防火墙穿透

WebRTC 采用独特的端到端媒体流，其中语音、视频和数据连接都直接在两个浏览器之间建立，详见第 5 章。遗憾的是，由于存在网络地址转换（Network Address Translation，NAT）和防火墙，增加了这一技术的实施难度，需要采用特殊的协议和过程才能实现。对等媒体有时还称为“端到端媒体”。

9.1 穿透简介

NAT 这种特性增加了建立直接端到端会话的难度。但是，通过利用一种名为“穿透”[BRYAN] 的技术，在许多情况都可以成功实现上述目的，平均成功率大概高达 85%。不过，这只是针对许多网络中的众多用户得出的平均结果。有些网络中的成功率会高一些，有些则低很多。例如，根据报告，美国移动数据网络中的成功率就比较低，大约为 30%。

采用穿透技术需要满足多项前提条件。这些条件包括：

1）尝试建立直接连接的两个浏览器必须同时发送“穿透”数据包。因此，二者必须都知道将要建立的会话，以及要将数据包发送到哪些地址。请注意，穿透数据包没有任何特殊之处，它只是普通的 IP 数据包，发送此数据包的目的是确定是否可以通过 NAT 访问特定的目标地址。

2）两个浏览器需要尽可能多地获知可用于访问对方的潜在 IP 地址。这些地址通常被描述为“私有”（或 NAT 内部）IP 地址、“公共”（或 NAT 外部）IP 地址和中继地址，具体取决于隐私设置（详见第 13 章）。

3）需要一个具有公共 IP 地址（即不位于 NAT 之后）并因此可由两个浏览器访问的媒

体中继，用作万不得已时的访问途径。

4）必须采用端到端流。换言之，UDP 通信所展现出来的行为必须类似于 TCP 连接。

通过使用 Web 服务器来协调穿透，可以满足要求 1。也就是说，Web 服务器知道将在两个浏览器之间建立会话，因此可确保二者在大体同一时间开始穿透。

要求 2 可通过使用 NAT 会话遍历实用工具（Session Traversal Utilities for NAT，STUN）服务器（详见 5.3 节）来满足。虚拟专用网络（Virtual Private Network，VPN）连接可能会提供额外的地址。也可以使用其他 NAT 遍历协议，例如通用即插即用（Universal Plug and Play，UPnP），不过这类协议并不常见。

要求 3 可通过使用中继型 NAT 遍历（Traversal Using Relay around NAT，TURN）服务器（详见 5.4 节）来满足。媒体中继地址是一个公共 IP 地址，用于转发从设置中继地址的浏览器收到的数据包，或者将收到的数据包转发给该浏览器。此地址随后会被添加到候选项列表中。

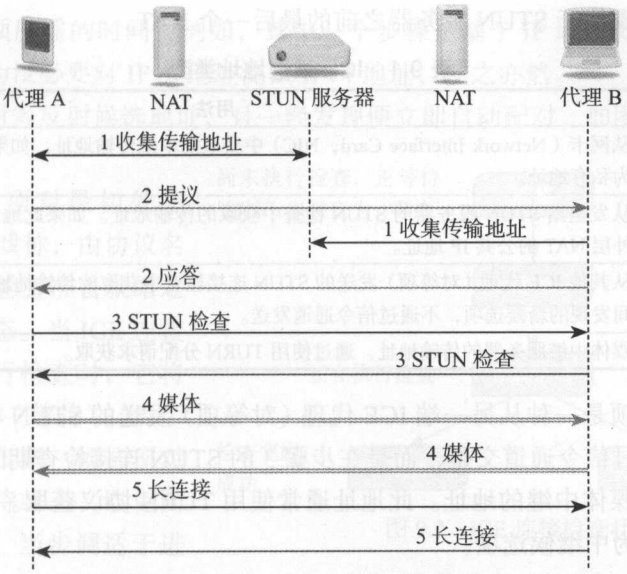
通过让浏览器使用其用来侦听传入媒体的同一 UDP 端口发送媒体，可以满足要求 4。这可以使 NAT 将 UDP 上的两个单向 RTP 会话识别为一个双向 RTP 会话。有关对称 RTP 的信息，请参见 12.3.2 节。

通过 TURN 服务器提供中继的媒体

在大多数情况下，通过穿透可最终建立直接对等连接。但是，在某些情况下，NAT 或防火墙的限制非常严格，无法建立直接路径，只有通过 TURN 服务器的地址才能成功。这样便产生了通过 TURN 服务器中继媒体的做法，如图 5.9 所示。在该图中，虽然 TURN 服务器显示为独立的服务器，但实际上 TURN 服务器是增加了中继功能的 STUN 服务器，在许多情况下，二者是一体的。虽然所有 TURN 服务器都同时具备 STUN 功能，但并非每台 STUN 服务器都具有 TURN 功能。

9.2 交互式连接建立

交互式连接建立（Interactive Connectivity Establishment，ICE）是一种标准穿透协议。它利用 STUN 和 TURN 服务器来帮助端点建立连接。图 9.1 显示了 ICE 中的基本步骤。



传输地址是 IP 地址和端口号的组合

1）收集候选传输地址
2）在信令通道中交换候选项
3）执行连接检查
4）选择选定的对并启动媒体
5）发送长连接（Keepalive）请求任何一端检测到使用中的 IP 地址发生变化，都重新启动 ICE（返回步骤 1）

图 9.1 ICE 呼叫流概览

下面几节内容将分别介绍这些步骤。

9.2.1 收集候选传输地址

第一步是收集候选传输地址。候选地址是或许可用于接收媒体以建立对等连接的 IP 地址和端口。在许多情况下，这些地址必须在呼叫时收集，而不能提前收集。在图 9.1 显示的示例中，一旦 A 处的用户发起与 B 建立对等连接的请求，ICE 代理 A 就会开始收集候选地址。一旦在信令通道中收到 A 发来的对等连接请求，ICE 代理 B 就会开始收集候选地址。

候选地址分为四种类型，表 9.1。主机候选地址是通过操作系统收集的地址，表示网卡（Network Interface Card，NIC）上的实际地址。如果 ICE 代理位于 NAT 之后，此地址就是私有 IP 地址，无法在子网之外路由到。接下来的两种候选地址称为“反射地址”，因为它们表示由 STUN 检查反射回 ICE 代理的地址，就如同客户端在通过 STUN 服务器照镜子，以此来识别自身的实际 IP 地址。服务器反射候选项是一种从响应中识别的地址，该响应针对的是发送给 STUN 服务器的 STUN 检查请求。如果 ICE 代理位于 NAT 之后，此地址将为最外层 NAT 的外部地址。换句话说，ICE 代理和 STUN 服务器之间可能存在多个 NAT 层，

但此项检查只能发现位于 STUN 服务器之前的最后一个 NAT。

表 9.1 ICE 候选地址类型

候选项类型	用法
主机候选项	从网卡（Network Interface Card，NIC）中获取的本地传输地址。如果此地址位于 NAT 之后，则为私有地址。
服务器反射候选项	从发送给 STUN 服务器的 STUN 检查中获取的传输地址。如果此地址位于 NAT 之后，则为最外层 NAT 的公共 IP 地址。
对等反射候选项	从其他 ICE 代理（对等项）发送的 STUN 连接检查中获取的传输地址。这是一种在连接检查期间发现的新候选项，不通过信令通道发送。
中继候选项	媒体中继服务器的传输地址。通过使用 TURN 分配请求获取。

对等反射候选项是一种从另一端 ICE 代理（对等项）发送的 STUN 检查中识别的地址。这种候选地址不通过信令通道交换，而是在步骤 3 的 STUN 连接检查期间发现。

中继候选项是媒体中继的地址。此地址通常使用 TURN 协议获取。使用 TURN 分配请求获取的传输地址为中继候选项。

对于在收集候选项这一步骤中使用的 STUN 和 TURN 服务器，将在浏览器中进行相应的配置。这通过 RTCConfiguration 对象的 iceServers 属性中的 STUN URI [RFC7064] 和 TURN URI [RFC7065] 实现。请注意，在大多数情况下，能够访问 TURN 服务器还意味着可使用 STUN 服务器功能。

需要指出的是，仅使用 STUN 识别公共 IP 地址候选项本身并不足以遍历 NAT。NAT 非常复杂，并因网络和服务提供商而存在很大差异。因此，需要完整的 ICE 功能才能确保实现 NAT 遍历。

9.2.2 交换候选项

第二步是通过信令通道交换候选地址。浏览器之间的候选项交换通过信令通道进行，详见第 4 章。一旦收到候选项，就会先对它们进行排序或确定优先级。一般而言，主机候选项的优先级最高，其次是反射地址，最后是中继候选项。如果在 IPv4 和 IPv6 之间有所偏好，则可通过不同的优先级设置来表示。候选项与 SDP 中的一个特定媒体流相关联。WebRTC 的默认行为是通过同一传输地址对所有媒体进行多路传输，包括语音、视频和数据。因此，只需一组候选项即可。

9.2.3 STUN 连接检查

一旦 ICE 代理发送并收到了候选项，就会开始进行连接检查。在图 9.1 中，对于代理 A 而言，当从代理 B 收到 SDP 应答时，便开始此项检查。对于代理 B 而言，在向代理 A 发送了 SDP 应答时，便开始此项检查。在此阶段，对于对等项发来的通过身份验证的任何 STUN 连接请求，ICE 代理都会生成 STUN 响应。

通过候选项配对和分析过程，可最大限度地减少执行的连接检查数量。这可以缩短

ICE 获取有效候选项所需的时间。例如，其中一个步骤是基于 IP 地址类型（IPv4 或 IPv6）对候选项配对，因为没必要对 IPv6 地址测试 IPv4 地址，反之亦然。

此步骤可发现对等反射候选地址，且一经发现便立即自动配对。如图 9.2 所示，存在五种可能的连接状态。

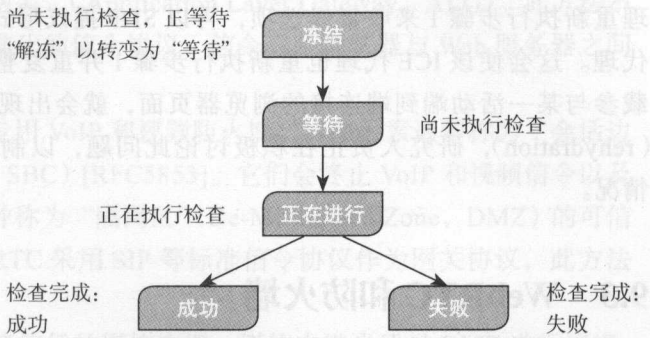


图 9.2　ICE 连接检查状态机

排队后的候选项对最初处于“冻结”状态（一种戏称，由协议名称引申而来），即在检查准备就绪之前一直处于待命状态。当 ICE 连接检查算法确定应执行检查时，它将“解冻”且状态变为“等待”。为了给检查创造均匀的步调，候选项对可能长时间处于等待状态，以免同时发送大量数据包。当步调适于进行检查时，一旦将 STUN 连接检查发送至另一端的对等端，状态将变为“正在进行”。如果有响应被发回，状态将变为“成功”，如果检查超时且没有响应，状态将变为“失败”。

有一种 ICE 优化技术称为“缓慢型 ICE”（Trickle ICE）（详见 11.5.1 节），它不是在 ICE 处理伊始便提供所有候选项，而是以尽可能小的候选项集合来启动 ICE，再随着处理的深入不断添加（缓慢加入）其他候选项。这些新候选项将被配对并排队，并经历图 9.2 中显示的相同步骤。目前，仅 Chrome 支持缓慢型 ICE。

9.2.4　选择选定的对并启动媒体

连接检查将一直持续，直至所有可能的检查都已完成（所有检查的状态都从“冻结”变为“成功”或“失败”）或者某一对候选项被选中。选择对的操作由施控 ICE 代理执行。ICE 协议有一种算法用于选择哪个浏览器是施控 ICE 代理，哪个浏览器是受控 ICE 代理。当收到的 STUN 连接检查的属性指示将使用某个候选项对时，受控 ICE 代理便可识别到另一端的 ICE 代理已选择该候选项对。随后，受控 ICE 代理将回复连接检查，确认将使用该对。现在，两个浏览器即可使用所选的候选项对发送媒体。

9.2.5　长连接

为确保 NAT 映射和过滤规则不在媒体会话期间超时，ICE 会不断通过使用中的候选项对发送连接检查，每隔 15 秒发送一次。这可以确保，即使媒体暂停或因其他情况而没有发送，也有数据包得到持续发送。如果媒体会话仍然处于活动状态，另一端的 ICE 代理就会生成 STUN 响应。如果这一端的 ICE 收到此 STUN 响应，即表明可继续发送媒体。如果没有收到 STUN 响应，则停止媒体流。这时需要重新启动 ICE，详见下一节。请注意，此行为没有在原始 ICE 协议规范中定义，而是在 ICE 扩展协议中定义（详见 11.3.10 节）。

9.2.6 ICE 重新启动

无论任何一端的 ICE 代理检测到传输基地址发生更改，都会触发重新启动 ICE 的事件。前面提到，基地址是用于生成正在使用的候选项对的传输地址。此事件会导致 ICE 代理重新执行步骤 1 来收集候选项，再以 SDP 提议形式将这些候选项发送给另一端的 ICE 代理。这会使该 ICE 代理也重新执行步骤 1 并重复整个过程。请注意，如果用户重新加载参与某一活动端到端连接的浏览器页面，就会出现这一情况。此过程有时称为“复水”（rehydration），研究人员正在积极讨论此问题，以制定相应的标准来最有效地处理这一情况。

9.3 WebRTC 和防火墙

防火幕墙用于防止建筑物和车辆中的火或热量四下传播；“防火墙”一词便由此而来，此事物在 Internet 中也十分常见。防火墙用于在网络边界上强制实施安全策略。它们可以实施任何类型的策略，并执行任何类型的 IP 数据包过滤或拦截。在大多数情况下，防火墙实施简单的数据包过滤规则，与 NAT 中使用的过滤规则十分相似。防火墙充当“单向网关”，允许从网络“内部”访问“外部”（Internet），但会拦截来自 Internet 的任意通信，防止其进入网络内部。就本质而言，防火墙的作用就是允许网络内部发起的正常 IP 通信，但拦截来自 Internet 的恶意通信。

一种策略是只允许传出通信，只有当来自外部的数据包被认定为内部请求的有效响应时，才会对这些数据包放行。例如，如果 TCP 连接由网络内部的某个主机开通，则防火墙可能会允许建立此连接。一旦连接开通，数据包便可双向流动，直至连接关闭。然而，来自外部的 TCP 连接将被拦截。UDP 通信的处理难度更高，因为 UDP 数据包不是连接的一部分，而且没有信令指示 UDP 流的开始和停止。在 UDP 通信中，防火墙可能也允许传出 UDP 数据包，并且对于某些传入的 UDP 数据包，如果它们的发送目标是最近刚发出 UDP 数据包的主机，则也可能予以放行。请注意，此防火墙行为非常类似于大多数 NAT 为限制在公共与私有 IP 地址和端口之间使用映射而应用的过滤规则。某些防火墙会拦截所有 UDP 通信，但 DNS 查找例外。在这种情况下，通过 TCP 运行媒体可能是建立媒体流的唯一方式。

防火墙与 NAT 拥有不同的功能，但经常同时实施在同一设备中。大部分家用路由器和企业路由器都内置有防火墙功能。由于 IP 访问由操作系统处理，因此在 PC 本身中配备防火墙功能的设计日益普遍。

WebRTC 防火墙遍历

用于遍历 NAT 的打洞技术通常也适用于遍历防火墙。但是，某些防火墙设有更为严格的规则，从而导致穿透失败。某些防火墙甚至会不加区分地拦截所有 UDP 通信。（对于这些

情况，研究人员已进行了不少讨论，旨在为通过 TCP 甚至 HTTP 传输 SRTP 制定标准。）

目前，有许多不同的方法用于帮助当今的 VoIP 和视频通信遍历防火墙。一种方法是在防火墙中内置识别 VoIP 信令和媒体协议的功能，以便防火墙在媒体会话期间仅为媒体开通“针孔”。这种方法有时称为“应用层网关”（Application Layer Gateway，ALG）。此方法不适用于 WebRTC，因为它没有使用标准化的信令协议，信令只是浏览器与 Web 服务器之间通过 HTTP 交换的一部分数据。

另一种方法是采用防火墙信任的专用 VoIP 和视频防火墙。这些元素通常称为“会话边界控制器”（Session Border Controller，SBC）[RFC5853]。它们会终止 VoIP 和视频信令以及媒体通信，并应用策略。SBC 使用一种称为“隔离区”（De-Militarized Zone，DMZ）的可信连接与防火墙相连。同样，除非 WebRTC 采用 SIP 等标准信令协议作为网关协议，此方法才有效。

一种可以使用的方法是利用防火墙信任的媒体中继。媒体中继也通过 DMZ 进行连接，并负责对流进行身份验证。TURN 服务器可提供此功能，且兼容 WebRTC 中使用的 ICE 打洞技术。基本而言，希望控制和监视 WebRTC 媒体流的企业会实施防火墙策略来导致所有穿透候选项（企业 TURN 服务器除外）失败。因此，所有流都由 TURN 服务器进行身份验证和中继。在 Web 浏览器中，可按照当前配置 Web 代理的方式（并出于同样的目的）配置 TURN 服务器。

防火墙还可以实施 ICE ALG。这可使防火墙将 ICE 打洞用作将有 UDP 流传入的信令。只要 ICE 长连接数据包不断发来，针孔就保持开放。

图 9.3 显示了使用 TURN 服务器通过防火墙的媒体。

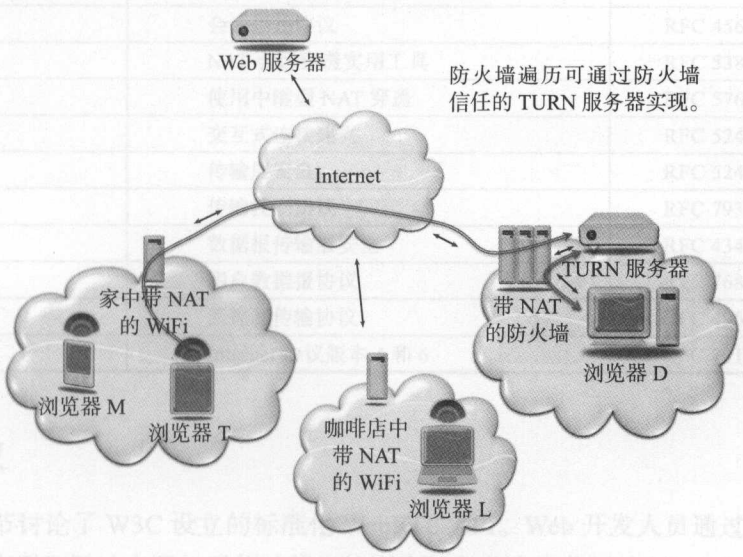


图 9.3　由防火墙遍历 TURN 服务器提供中继的媒体

更多关于媒体数据穿越防火墙的信息，可查看 [IEEE-COMS] 以及 [draft-hutton-rtcweb-nat-firewall-considerations]。

9.4 参考资料

[BRYAN] http://www.brynosaurus.com/pub/net/p2pnat

[RFC7064] http://tools.ietf.org/html/rfc7064

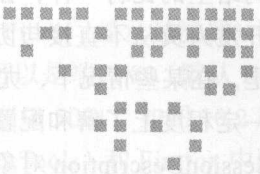
[RFC7065] http://tools.ietf.org/html/rfc7065

[RFC5853] http://tools.ietf.org/html/rfc5853

[IEEE-COMS] Alan Johnston、John Yoakum 和 Kundan Singh 共同编写的“Taking on WebRTC in an Enterprise”（在企业中采用 WebRTC），《IEEE Communications Magazine》（《IEEE 通信杂志》），第 4 期第 51 卷，2013 年 4 月。

[draft-hutton-rtcweb-nat-firewall-considerations]

http://tools.ietf.org/html/draft-hutton-rtcweb-nat-firewallconsiderations



第 10 章 Chapter 10

协　议

目前，存在许多与 WebRTC 相关的协议。表 10.1 列出了最重要的一些协议。本章将讨论这些协议的用途。具体的协议体系结构，请参见图 10.1。

表 10.1　WebRTC 协议

协议	用途	规范
HTTP	超文本传输协议	RFC 2616
WebSocket	Web 浏览器与服务器之间的套接字	RFC 6455
SRTP	安全实时传输协议	RFC 3711
SDP	会话描述协议	RFC 4566
STUN	NAT 会话穿透实用工具	RFC 5389
TURN	使用中继型 NAT 穿透	RFC 5766
ICE	交互式连接建立	RFC 5245
TLS	传输层安全	RFC 5246
TCP	传输控制协议	RFC 793
DTLS	数据报传输层安全	RFC 4347
UDP	用户数据报协议	RFC 768
SCTP	流控制传输协议	RFC 4960
IP	Internet 协议版本 4 和 6	RFC 791、RFC 2460

10.1　协议

前面的章节讨论了 W3C 设立的标准化 WebRTC API。Web 开发人员通过直接使用这些 API 在他们的应用和网站中添加通信功能。接下来的几节内容将讨论 WebRTC 使用的协议。

这些协议就像网络上的比特一样，让浏览器之间以及浏览器和服务器之间相互通信。一般情况下，Web 开发人员从不直接与协议交互，因为协议的默认设置和配置通常可以满足他们的需要。但是，在某些情况下，尤其是在 WebRTC 客户端与非 WebRTC 客户端进行通信时，就需要在一定程度上了解和配置 WebRTC 使用的协议。此外，如果需要调整 WebRTC API 中的 RTCSessionDescription 对象（例如当用户代理之间存在 SDP 互操作性问题时，就可能需要如此），应用程序作者将需要深入了解协商的工作机制。无论在何种情况下，大致了解 WebRTC 使用的协议都有助于开发人员开展工作；本章就介绍这些基本信息。另一方面，希望利用 WebRTC 的电话开发人员需要详细了解其中使用的协议。对于这些人员来说，后续章节详细介绍了这些协议如何协同工作。

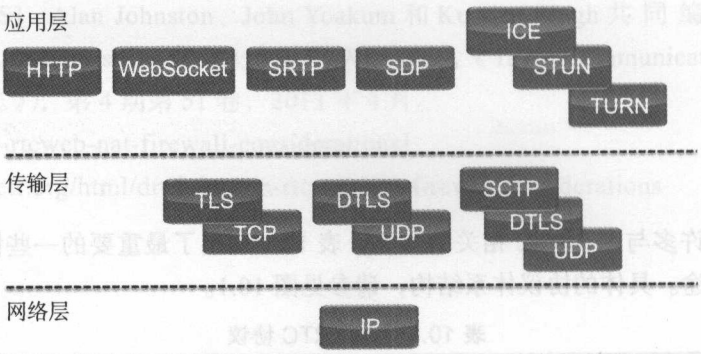


图 10.1　WebRTC 中的协议

Wireshark 说明

Wireshark [WIRESHARK] 是一款用于查看网络协议的开源工具。它非常有助于了解 WebRTC 的各种协议并进行故障排除。若要使用 Wireshark，最简便的方法是在运行参与对等连接的浏览器的计算机上运行它。以下几节说明了如何观察 WebRTC 的主要协议。这些示例和屏幕截图基于 Wireshark 1.10.5 Mac 版。

10.2　WebRTC 协议概述

10.2.1　HTTP 协议

WebRTC 无疑会使用超文本传输协议（Hyper-Text Transport Protocol，HTTP）[RFC2616]。它是万维上浏览器和服务器之间通信的协议。WebRTC 使用 HTTP 的方式与其他 Web 应用程序一样，不需要其他特殊的知识。HTTP 的最新版本为 1.1。目前，IETF 正在制定 HTTP 的下一版本，称为 2.0。此协议有可能提高 Web 下载操作和应用程序运行的速度和效率。WebRTC 将能够使用当前版本及所有未来版本的 HTTP。有关如何使用 HTTP

建立 WebRTC 信令通道的示例，请参见 4.3.3 节。

通过 Wireshark 可监视 HTTP，方法是将“Display Filter”（显示过滤器）设置为“http”，或者在 TCP 过滤器中设置用于 HTTP 的端口。该端口可以是端口 80（默认 HTTP 端口），也可以是其他端口。在本书的演示代码中，我们使用了端口 5001，如图 10.2 所示。请注意，Chrome 中的“工具 / 开发人员工具”（Tools / Developer Tools）和 Firefox 中的“工具 / Web 开发者”（Tools / Web Developer）菜单可用于检测浏览器中的 HTTP。

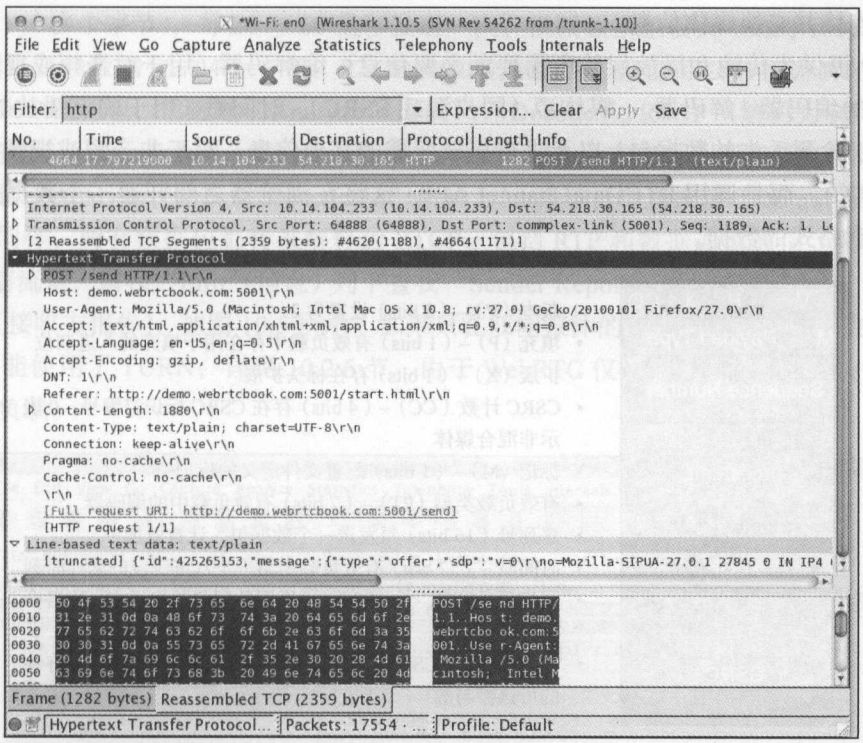


图 10.2 Wireshark 捕获的 HTTP 信令（SDP）数据

10.2.2 WebSocket 协议

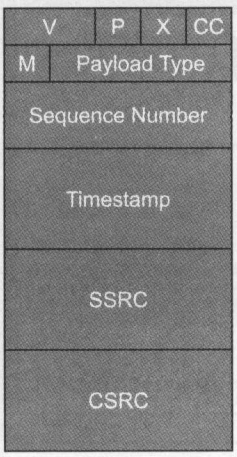
WebSocket 协议 [RFC6455] 允许浏览器打开新的与 Web 服务器双向通信的连接。此连接使用 HTTP 信令机制开通，拥有与 HTTP Web 会话类似的安全特性，可复用已有的 HTTP 连接。这可以避免在浏览器与 Web 服务器之间进行 HTTP 轮询和开通多个 HTTP 连接。在打开连接的时候，浏览器使用 WebSocket 向浏览器发出指示，这称为“WebSocket 子协议”。示例包括 SIP WebSocket 子协议或 XMPP WebSocket 子协议，详见 4.3.6 节和 4.3.7 节。

Wireshark 可用于检测 WebSocket 通信，方法是将“Display Filter”（显示过滤器）设置为“http”，然后查找 HTTP/1.1 “101 Switching Protocols”（101 交换协议）响应。请注意，Wireshark 目前无法直接对 WebSocket 通信内容解码。

10.2.3 RTP 协议和 SRTP 协议

WebRTC 使用的最重要的协议是实时传输协议（Real-time Transport Protocol，RTP）[RFC3550]。WebRTC 仅使用 RTP 或安全 RTP（Secure RTP，SRTP）[RFC3711] 的安全配置文件。SRTP 协议用于在 WebRTC 客户端之间传输音频和视频媒体数据包。媒体数据包中含有由麦克风、摄像头或应用程序生成的数字化音频帧或视频帧，并使用扬声器或显示器播放。成功建立对等连接并完成提议 / 应答交换后，将在浏览器之间或浏览器与服务器之间建立 *SRTP 连接并交换媒体信息*。

SRTP 提供为传输和播放媒体所需要的必要信息：编解码器（用于对音频或视频进行采样和压缩的编码器 / 解码器）、媒体源（同步源或 SSRC）、时间戳（用于确定播放时刻）、序列号（用于检测丢失的数据包）以及进行播放所需的其他信息。对于非音频或视频数据，将不使用 SRTP，而是调用 RTCDataChannel API，这将在浏览器之间开通一个数据通道，以便交换任意格式的数据。



- 版本（V）–（2 bits）设置为 2
- 填充（P）–（1 bits）有效负载中存在用于填充的八进制位
- 扩展（X）–（1 bits）存在标头扩展
- CSRC 计数（CC）–（4 bits）存在 CSRC 标识符编号；0 表示非混合媒体
- 标记（M）–（1 bits）配置文件定义的标记
- 有效负载类型（PT）–（7 bits）有效负载中的编码器
- 序列号（16 bits）每发送一个数据包，计数就加一
- 时间戳 –（32 bits）有效负载中第一个八进制位的采样时刻
- 同步源（SSRC）–（32 bits）标识数据包的发送方，在各会话中均不相同
- 参与 SSRC（CSRC）–（32 bits）有效负载中因混合而存在的可选参与源

图 10.3 RTP 头部

表 10.2 显示了 WebRTC 使用的 RTP 扩展集合。

表 10.2 WebRTC RTP 扩展摘要

扩展	说明	参考
SAVPF 配置文件	带反馈的音视频安全配置文件，使用安全 RTP 和早期 RTCP 反馈	12.1.4 节
RTP/RTCP 传输复用	RTP/RTCP 在相同 UDP 端口上接收	12.1.5 节
音视频传输复用	所有媒体都通过同一 UDP 端口接收，媒体类型（音频或视频）由 SSRC 和有效负载类型标识	11.5.4 节
对称 RTP	通过同一 UDP 端口收发 RTP 媒体，以简化 NAT 穿透	12.3.2 节
拥塞避免和控制	通过停止发送 RTP 或调整带宽使用来避免拥塞	11.5.5 节

Wireshark 无法自动解析 RTP。一般情况下，只将“Display Filter”（显示过滤器）设置

为“rtp”不会显示任何RTP数据包。正确做法是在先将“Display Filter”设置为“udp”以找出UDP通信流量。建立对等连接后，浏览器之间将定期发送UDP通信。在基于UDP的RTP通信中，第一个8比特是十六进制数90（如果不存在RTP标头扩展项，则为十六进制数80）。通过选择这些UDP数据包之一，再选择“Analyze”（分析）/“Decode As”（解码为），然后选择“RTP”，Wireshark就会将这些数据包作为RTP处理，如图10.4所示。通过检查SDP，将可以确定音频和视频的有效负载类型（Payload Type，PT）。音频数据包较小（只有100多个字节），每20～60毫秒发送一次。视频数据包较大（大约1000个字节），发送频率是音频数据包的4～5倍。RTCP和数据通道数据包也可以解析为RTP数据包。要查看RTCP数据包，请选择“Analyze”（分析）/“Decode As RTCP”（解码为RTCP）。（如果你已经执行“Decode As”（解码为）操作，例如以RTP形式查看，则需要先清除此设置，方法是选择“Analyze”（分析）/“Decode As”（解码为），再单击单选按钮“Do not decode”（不解码）。）同样，所有数据包都将被解释为RTCP，包括RTP和数据通道通信。要查找实际RTCP通信流量，请在“Info”（信息）列下查找“Sender Report”（发送方报告）或“Receiver Report”（接收方报告），如图10.5所示。如果只在双向媒体的一个方向上找到RTP数据包，则说明可能使用了TURN，详见10.2.6节。由于WebRTC仅使用SRTP，将无法解密RTP媒体有效负载。

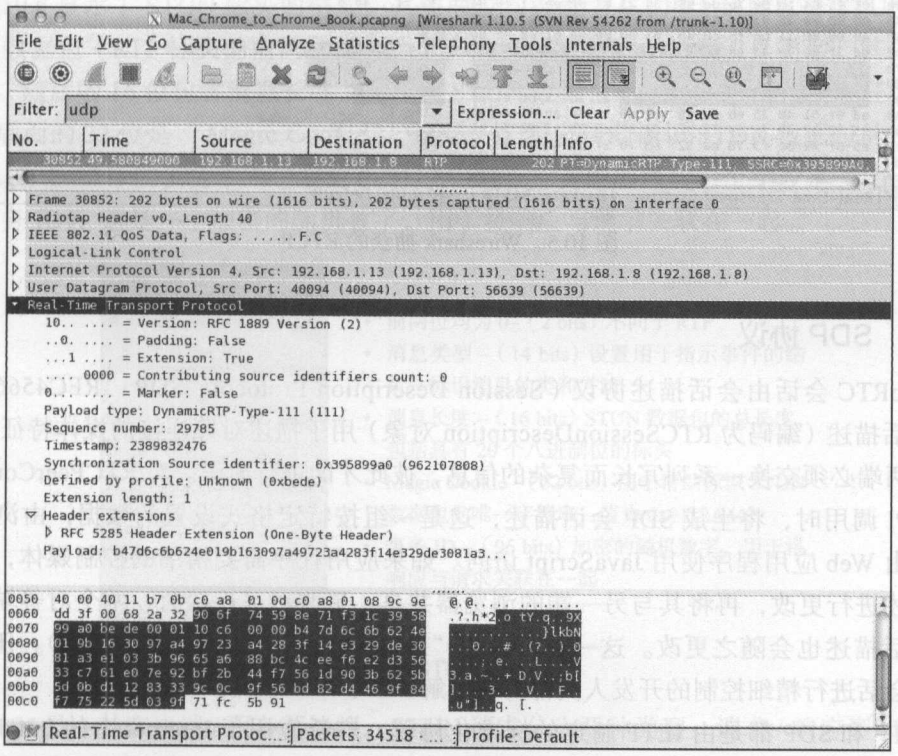


图10.4 Wireshark捕获的RTP

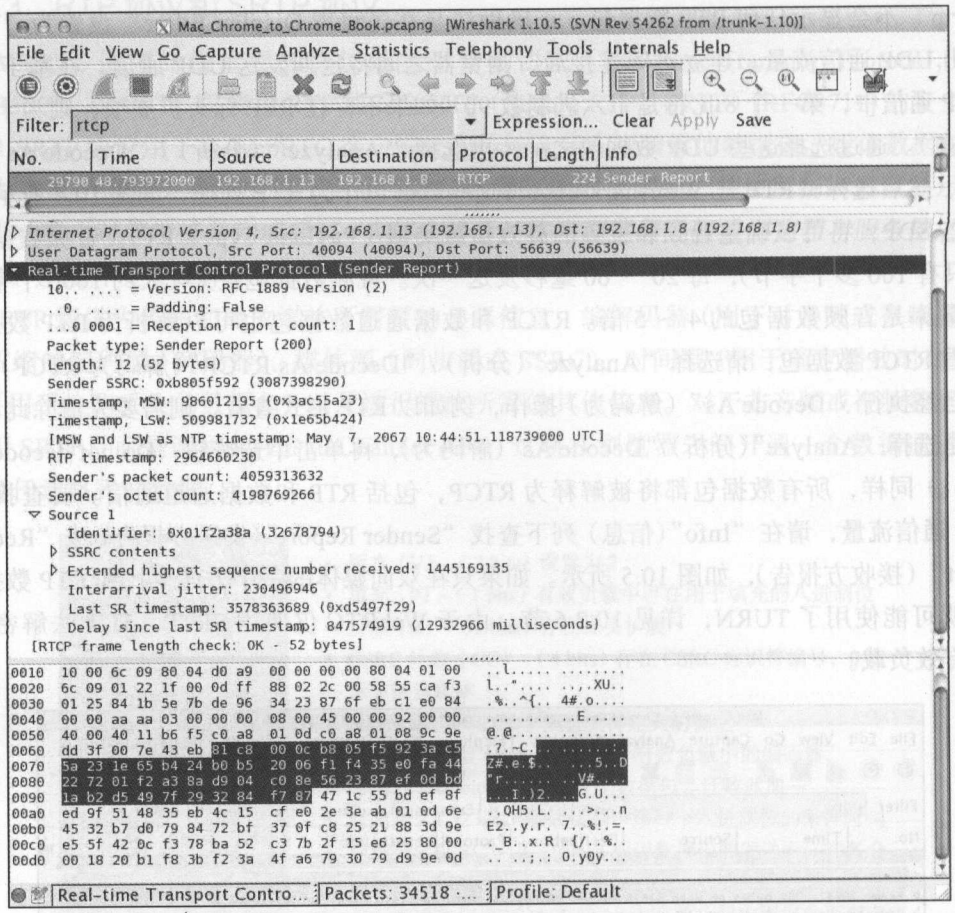


图 10.5 Wireshark 捕获的 RTCP

10.2.4 SDP 协议

WebRTC 会话由会话描述协议（Session Description Protocol，SDP）[RFC4566] 描述。SDP 会话描述（编码为 RTCSessionDescription 对象）用于描述对等连接的媒体特征。SRTP 会话的两端必须交换一系列冗长而复杂的信息，彼此才能进行通信。对 RTCPeerConnection 进行 API 调用时，将生成 SDP 会话描述，这是一组按特定格式设置的数据，由浏览器生成，并由 Web 应用程序使用 JavaScript 访问。如果应用程序需要精细的控制媒体，可先对会话描述进行更改，再将其与另一端的浏览器共享。更改对等连接时，两个对等端之间交换的会话描述也会随之更改。这一过程称为“提议 / 应答”交换（详见 6.2 节）。任何希望对媒体会话进行精细控制的开发人员都需要了解 SDP。

SRTP 和 SDP 都是由 IETF 制定的标准化协议，广泛应用于 Internet 上的 Internet 通信设备和服务中，例如 IP 语音（Voice over IP，VoIP）电话、客户端、网关，以及视频会议和

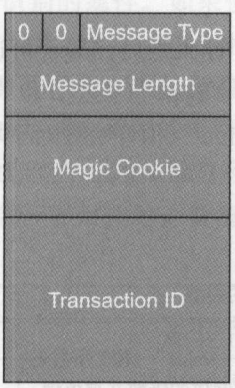
协作设备。这样上面提到的任一设备或客户端与 WebRTC 客户端通信就存在可能性了。但是，当今很少有 VoIP 或视频设备或客户端支持 WebRTC 的全套功能和协议。为支持这些新协议，需要升级上述设备或者在 WebRTC 客户端与 VoIP 或视频客户端之间利用网关功能执行转换。电话和 Internet 通信开发人员可使用本书中有关这些协议的说明来指导自己开发兼容 WebRTC 的客户端或网关。

浏览器之间的 SDP 提议 / 应答交换将通过信令通道发送。如果 Wireshark 捕获的信令通道没有加密，则可以检查 SDP，详见图 10.2 中的示例。本书中的示例应用程序采用另一种方式，即在视频窗口下的窗口中显示 SDP 提议和应答，如图 6.3 所示。

10.2.5 STUN 协议

NAT 会话穿透实用工具（Session Traversal Utilities for NAT，STUN）[RFC5389] 协议用于帮助进行 NAT 穿透。在 WebRTC 中，STUN 客户端将内置在浏览器用户代理中，STUN 服务器则由 Web 服务器运行。在会话建立之前，先发送 STUN 测试报文，以便浏览器确定其是否位于 NAT 之后并发现映射地址和端口。这些信息随后将用于构建在 ICE 打洞时使用的候选地址。STUN 可基于 UDP、TCP 或 TLS 传输。STUN 端口号可使用 DNS SRV 查询操作来确定；STUN 的默认 UDP 端口为 3478。

图 10.6 显示了 STUN 标头的格式。其前两位均设置为 0，用于帮助区分 STUN 数据包和 RTP 数据包（RTP 标头的前两位设置为 0 和 1）。接下来 14 位是消息类型，其中包括类和方法。随后的 16 位包含消息长度，即整个 STUN 数据包的大小，包括标头和任何填充内容。再后面的 32 位是“Magic Cookie”，这是一个用来帮助标识 STUN 数据包的唯一字符串。Magic Cookie 的值为十六进制数 0x2112A442。标头中的最后一个字段是“Transaction ID”（事务 ID），这是一个加密的随机数字，用于将响应与请求关联在一起。



- 前两位均为 0–（2 bits）不同于 RTP
- 消息类型 –（14 bits）设置用于指示事件的结束—标识消息的类和方法
- 消息长度 –（16 bits）STUN 数据包的总长度，包括具有 20 个八进制位的标头
- Magic Cookie –（32 bits）用于帮助标识 STUN 数据包的唯一字符串，值为 0x2112A442
- 事务 ID –（96 bits）加密的随机数字，用于将响应与请求关联在一起

图 10.6 STUN 标头格式

STUN 是一种客户端 / 服务器协议。STUN 请求分为两种类型：请求 / 响应型和指示型。在请求 / 响应型请求中，发起请求的 STUN 客户端需要从收到请求的 STUN 服务器获得响

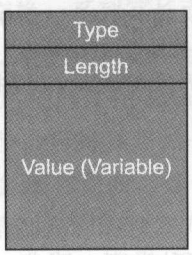
应，因此必要时将会重新发送请求，确保收到响应。当基于 UDP 发送此类请求时，将使用一种名为“重新传输超时”（Retransmission Time Out，RTO）的指数后退定时器重新传输请求 / 响应型请求消息。可以对 RTO 进行配置，但必须大于 500 毫秒。当 RTO 到期时，将进行第 1 次重新传输；当 RTO 第 2 次到期时，将进行第 2 次重新传输；当 RTO 第 3 次到期时，将进行第 3 次重新传输，以此类推。当发送的请求总数达到 7 次后，将结束重新传输。在指示型请求中，不要求提供响应。

在核心 STUN 协议中，只有一种称为“ Binding”的方法。STUN 客户端使用 Binding 来创建和发现 NAT 映射。如果 STUN 客户端位于 NAT 之后，而 STUN 服务器位于 NAT 之外，则发送 STUN Binding 请求将导致 NAT 创建映射并分配公共 IP 地址和端口。同时，这会在 NAT 中创建一条过滤规则，规定谁可以使用此映射向内网发送数据包。当 STUN 服务器收到 STUN Binding 请求时，它会记录 STUN Binding 请求来自哪个 IP 地址和端口号。此地址和端口号随后将以 STUN Binding 响应的形式返回给客户端。收到 STUN Binding 响应的 STUN 客户端会将响应中发来的 IP 地址和端口与它用于发送请求的 IP 地址和端口进行比较。如果二者相同，则说明客户端和服务器之间没有 NAT。如果二者不同，则说明至少有一个 NAT，而且客户端能够识别由最外层的 NAT 分配的 IP 地址和端口。请注意，STUN 客户端和服务器之间可能存在多个 NAT，但只能识别最外层 NAT 的相关信息。

客户端需要按固定间隔以指示形式发送 STUN Binding 请求，防止 NAT 映射超时。指示型请求可使 NAT 重置其 UDP 定时器。只要 STUN Binding 指示的发送间隔短于客户端和服务器之间的最短 NAT UDP 定时器设置，就可以保持 NAT 映射。

STUN 有两种身份验证机制：短期身份验证和长期身份验证。短期身份验证采用用户名 / 密码，此机制适用于单个会话。ICE 使用此方法对每组连接检查应用不同的身份验证。长期身份验证采用质询 / 响应机制，允许使用长期凭据。例如，可重复使用 SIP 身份验证凭据。

图 10.7 显示了 STUN 属性的格式。属性位于 STUN 标头之后，属于可选信息。属性编码为“类型、长度和值”(Type,Length,and Value,TLV)。“Type”(类型) 和 “Length”(长度) 长为两个八进制位，但 “Value”(值) 的长度可变。“Value”(值) 始终处于填满状态，因此其长度是 4 个字节的倍数。



- 类型 –（16 bits）属性的类型
- 长度 –（16 bits）属性的总长度
- 值 –（长度可变）所有 “ Value”（值）字段的长度都必须是 4 个八进制位的倍数并被填满

图 10.7　STUN 属性格式

表 10.3 列出了核心协议中的 STUN 属性。在 Binding 响应中，由 STUN 识别的 IP 地址和端口号将通过 XOR-MAPPED-ADDRESS 属性返回。XORMAPPED-ADDRESS 属性会对 IP 地址和端口号执行异或（exclusive OR）运算，以实现模糊处理。这是因为，某些 NAT 具有一种名为常规应用层网关（Application Layer Gateway，ALG）的功能，该功能会查找与 NAT 映射关联的 IP 地址和端口并动态重写这些信息。（当存在多种协议时，此类 ALG 可能会导致严重问题，应尽量在网络中予以禁用或删除。）如果服务器同时包括这两个属性，并且客户端在这两个属性中识别的信息并不相同，则可以断定存在此类 ALG。MAPPED-ADDRESS 属性仅适用于需要向后兼容原始 STUN 协议的情形。

表 10.3 STUN 属性

属性	用途
XOR-MAPPED-ADDRESS	经过异或运算的客户端反射传输地址
MAPPED-ADDRESS	服务器识别的客户端反射传输地址
USERNAME	消息完整性功能中使用的用户名
MESSAGE-INTEGRITY	STUN 消息的加密消息身份验证代码
FINGERPRINT	STUN 消息的循环冗余代码（Cyclic Redundancy Code，CRC）
ERROR-CODE	出错响应的错误代码和原因描述
REALM	用于长期身份验证的作用域
NONCE	用于长期身份验证的 Nonce
UNKNOWN-ATTRIBUTES	420 出错响应中的未知属性
SOFTWARE	用于调试的 STUN 制造商和版本信息
ALTERNATE-SERVER	300 出错响应中的备选 STUN 服务器

失败响应由响应中的 ERROR-CODE 属性承载。表 10.4 显示了 STUN 错误代码集合。如果使用的代码为“420 Unknown Attribute”（420 属性未知），UNKNOWN-ATTRIBUTES 属性将列出未识别的属性。如果使用的代码为“300 Try Alternative”（300 请尝试其他操作），ALTERNATIVE-SERVER 属性将包含要使用的其他 STUN 服务器的 IP 地址和端口。

STUN 服务器没有实施任何身份验证机制。这是因为，对请求进行身份验证所需的工作量大于直接响应请求所需的工作量。

目前，新成立的 TURN 修订和革新（TURN Revised and Modernized，TRAM）工作组正在对 STUN 制定扩展内容。详情请参见 11.5.9 节。

表 10.4 STUN 错误代码

错误代码	原因描述	用途
300	Try Alternative（请尝试其他操作）	STUN
400	Bad Request（请求错误）	STUN
401	Unauthorized（未经授权）	STUN
403	Forbidden（禁止操作）	TURN
420	Unknown Attribute（属性未知）	STUN

（续）

错误代码	原因描述	用途
437	Allocation Mismatch（Allocation 不匹配）	TURN
438	Stale Nonce（Nonce 过时）	STUN
441	Wrong Credentials（凭据错误）	TURN
442	Unsupported Transport Protocol（传输协议不受支持）	TURN
486	Allocation Quota Reached（已达到 Allocation 配额）	TURN
487	Role Conflict（角色冲突）	ICE
500	Server Error（服务器错误）	STUN
508	Insufficient Capacity（容量不足）	TURN

通过将“Display Filter”（显示过滤器）设置为“stun”，可使 Wireshark 找出 STUN，如图 10.8 所示。但是，这会同时找到 STUN、ICE 和 TURN 数据包。通过检查存在的消息类型和属性，可将 STUN 数据包与 ICE 和 TURN 数据包区分开来。此外，要仅查找 STUN Binding 请求和响应，可在端口 3478 中进行过滤。ICE STUN 数据包将由用于 RTP 媒体的相同端口传输。本书中的演示应用程序默认使用 STUN 生成服务器反射候选地址。要关闭 STUN，请在 URL 中加入属性“stunuri=0”。

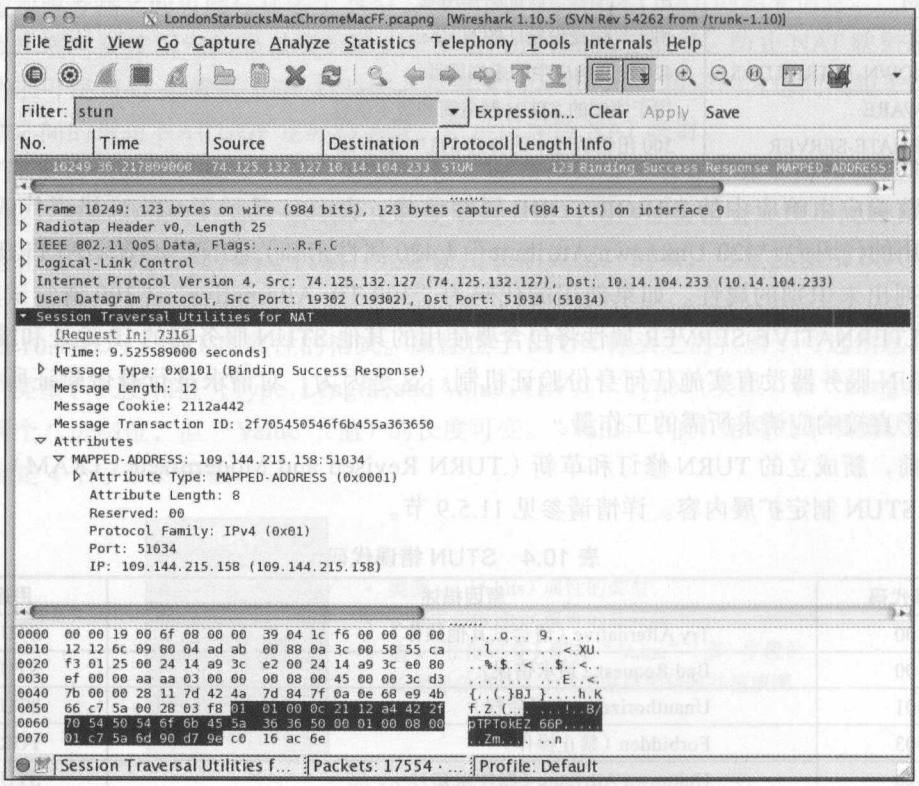


图 10.8 Wireshark 捕获的 STUN

10.2.6 TURN 协议

使用 TURN[RFC5766] 是对 STUN 协议的扩展，用于在 ICE 打洞失败时提供媒体中继。在 WebRTC 中，浏览器用户代理中包含一个 TURN 客户端和 Web 服务器，服务提供商或者企业会提供一个 TURN 服务器。浏览器请求一个来自 TURN 服务器的公有 IP 地址和端口号作为传输中继地址。此地址随后将作为候选地址纳入 ICE 打洞过程。TURN 还可用于穿透防火墙，详见 9.3 节。TURN 端口号可使用 DNS SRV 查找操作来确定；TURN 的默认 UDP 端口为 3478。

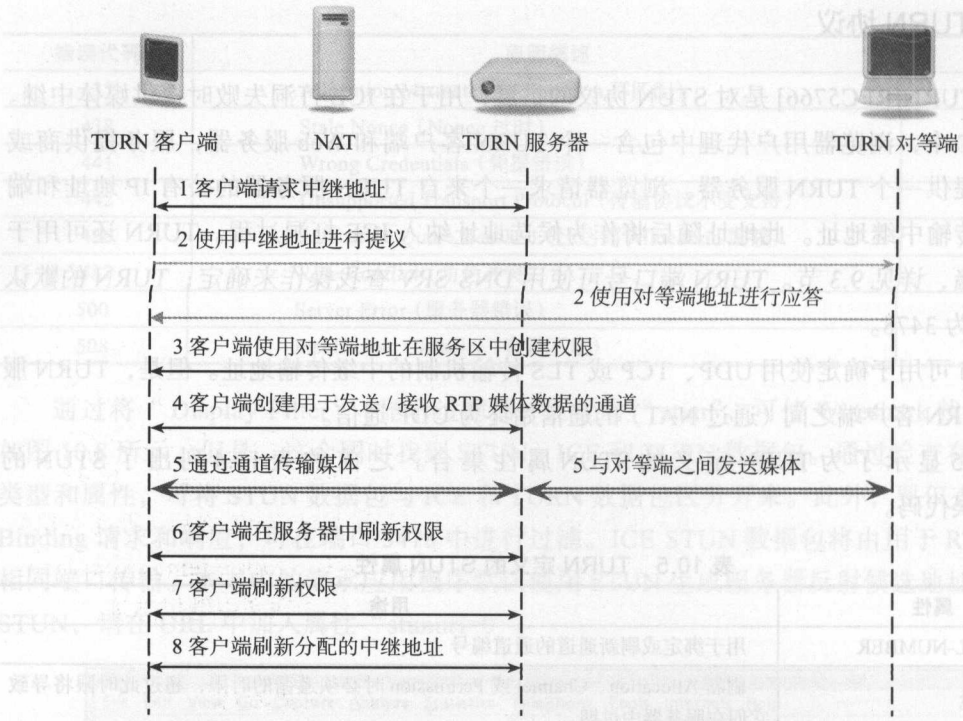
TURN 可用于确定使用 UDP、TCP 或 TLS 传输机制的中继传输地址。但是，TURN 服务器和 TURN 客户端之间（通过 NAT）的通信始终为 UDP 通信。

表 10.5 显示了为 TURN 定义的 STUN 属性集合。之前的表 10.4 列出了 STUN 的 TURN 错误代码。

表 10.5 TURN 定义的 STUN 属性

属性	用途
CHANNEL-NUMBER	用于绑定或刷新通道的通道编号
LIFETIME	刷新 Allocation、Channel 或 Permission 时必须遵循的时限，超过此时限将导致它们在服务器中过期
XOR-PEER-ADDRESS	有权使用中继地址的对等端的服务器反射地址
DATA	包含 Send 和 Data 中的中继应用程序数据
XOR-RELAYED-ADDRESS	在 Allocate 响应中分配的中继地址
EVEN-PORT	为 RTP 请求偶数端口号，并为 RTCP 请求下一个更高的端口号
REQUESTED-TRANSPORT	请求的中继方与对等端之间的传输机制，通常为 UDP
DONT-FRAGMENT	对发送至对等端的应用程序设置为不拆分位
RESERVATION-TOKEN	唯一标识中继传输地址

除了 Binding 之外，TURN 定义了一整套全新的 STUN 方法，如表 10.6 所示。Allocate 方法用于从 STUN 服务器请求中继传输地址。Refresh 方法用于刷新和保活已经分配好的地址。CreatePermission 方法用于对中继地址设置过滤规则（类似于 NAT 使用的规则）。TURN 有两种方式来通过 TURN 服务器中继数据。一种方式是使用 Send 和 Data 方法，此时中继数据由 STUN 消息承载。另一种方式是在 TURN 客户端和 TURN 服务器之间建立通道，并使用 ChannelData 消息发送数据，此消息不使用 STUN 标头，没有 36 个字节的开销。对于音频或视频媒体，通常会使用通道。建立通道时使用 ChannelBind 方法。



1）客户端从 TURN 服务器请求分配中继地址并在响应中接收此信息。
2）在提议中，客户端向对等端发送中继地址；在应答中，对等端向客户端发送对等端地址。
3）客户端使用服务器对等端地址在服务器中创建权限。
4）客户端创建用于发送和接收 RTP 媒体的通道。
5）客户端在通道中发送和接收媒体；对等端向 TURN 服务器发送媒体并接收来自该服务器的媒体。
6）客户端在服务器中刷新权限。
7）客户端刷新与服务器之间的通道。
8）客户端刷新服务器分配的中继地址。

图 10.9　TURN 呼叫流概览

表 10.6　STUN 方法

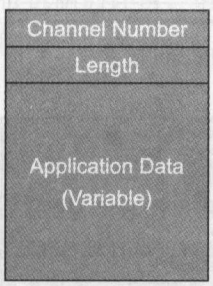
STUN 方法	用途	规范
Binding	创建并保持 NAT 映射	STUN
Allocate	客户端接收分配的中继传输地址	TURN
Refresh	定期对来自客户端的分配发送长连接请求	TURN
Send	从客户端向服务器发送的应用程序数据（不用于 RTP）	TURN
Data	从服务器（对等端）向客户端发送的应用程序数据（不用于 RTP）	TURN
CreatePermission	在服务器中设置权限，以允许特定的 IP 地址使用中继地址	TURN
ChannelBind	创建或刷新用于交换 RTP 数据的通道	TURN

TURN 服务器通常使用 STUN 长期身份验证。对于 TURN 请求，可使用包含 REALM 和 NONCE 属性的“401 Unauthorized”（401 未经授权）响应进行质询。根据质询，TURN

服务器将使用 USERNAME 属性（包含用户的用户名）和 MESSAGE-INTEGRITY 属性（使用用户名、作用域和密码带密钥的 MD5 散列消息鉴别码 /HMAC）重新发送请求。这类似于 SIP Digest 和 HTTP Digest 的质询 / 响应身份验证，但又不完全一样。

有关 TRAM 工作组当前为 TURN 制定的新扩展内容，请参见 11.5.9 节。

通过将“ Display Filter”（显示过滤器）设置为“ stun”并查找 Allocate、Refresh 等方法（如表 10.6 所示），可在 Wireshark 中找出 TURN 数据包。图 10.11 中显示了 TURN Allocate 请求，图 10.12 中显示了 Allocate 响应。此外，还可基于 TURN 传输媒体。查找定期发送、标记为 ChannelData 消息的 STUN 数据包，如图 10.10 和图 10.12 所示。请注意，图 10.13 中的 TURN ChannelData 有效负载是一个完整的 RTP 数据包（以十六进制数 90 开头）。在默认情况下，本书的演示中不使用 TURN，但可通过“turnuri=1”URL 属性来启用它。请注意，目前只有 Chrome 支持 TURN，如果对 Firefox 启用 TURN，似乎会导致故障。例如，如果使用 URL“ http://demo.webrtcbook.com:5001/start.html?turnuri=1”，将添加一个 TURN 中继候选项。如果两个浏览器位于不同的网络上且都处于 NAT 之后（因此主机候选项无效），则要强制使用 TURN 服务器传送媒体，可使用 URL“ http://demo.webrtcbook.com:5001/start.html?turnuri=1&stunur i=0”，这时将只使用主机候选项和中继候选项。



- 通道编号 –（16 bits）CreateChannel 请求中的 CHANNEL-NUMBER 属性
- 长度 –（16 bits）Application Data（应用程序数据）字段的总长度（八进制位数）
- 应用程序数据 –（长度可变）对等端发送或接收的应用程序数据

图 10.10 TURN ChannelData 格式

10.2.7 ICE 协议

WebRTC 中使用的另一个关键协议是交互式连接建立（Interactive Connectivity Establishment，ICE）协议 [RFC5245]。ICE 有两项重要功能：

1）ICE 可使位于 NAT 设备后的 WebRTC 客户端互相交换媒体。

2）ICE 提供通信许可验证功能。这意味着，媒体数据包将只发送给应该收到该通信的浏览器。恶意 Web 应用程序可能会诱骗浏览器将数据发送给不是通信方的 Internet 主机。这种攻击称为拒绝服务（Denial of Service，DOS）洪泛攻击。ICE 可防止此类攻击得逞，因为如果不成功完成 ICE 交换，就绝不会发送媒体。

ICE 采用一种称为“打洞”的技术（详见 9.2 节），这种技术最先由网络游戏玩家使用，即便他们的 PC 位于 NAT 之后，他们也需要直接交换数据包，以开展多人游戏。会话开始时，先运行 ICE，之后才在浏览器之间建立 SRTP 会话。它还用于建立非媒体数据通道。

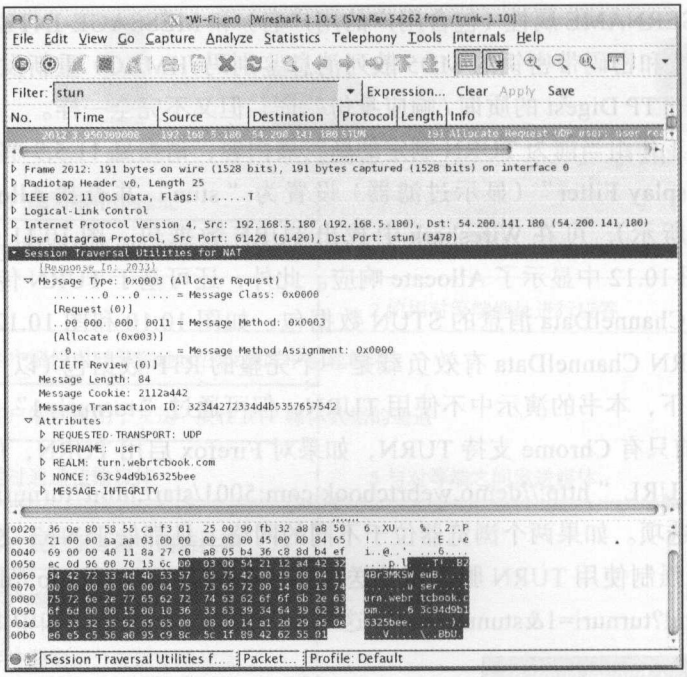


图 10.11 Wireshark 捕获的 TURN Allocate 请求

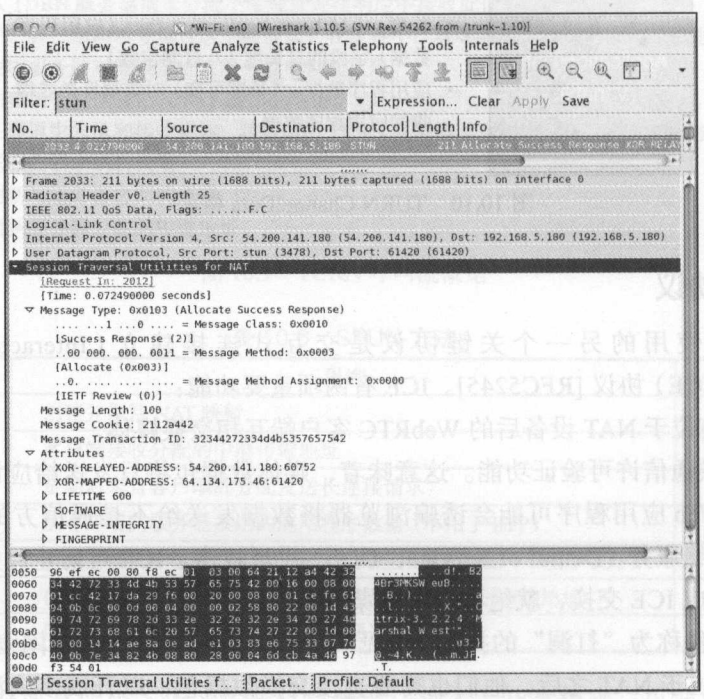


图 10.12 Wireshark 捕获的 TURN Allocate 响应

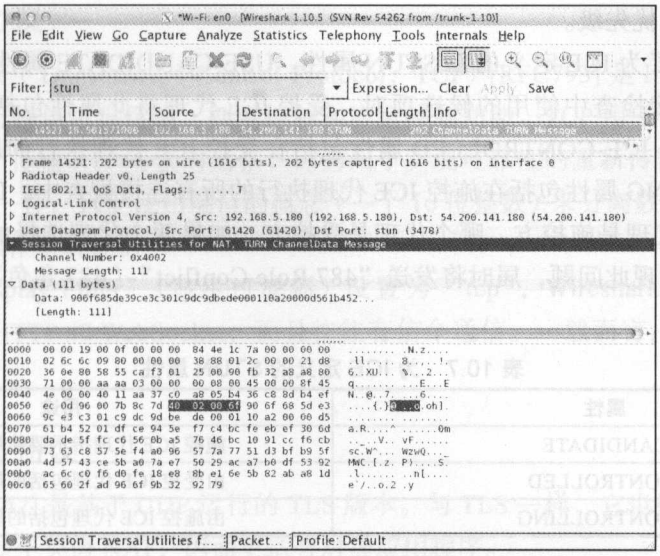


图 10.13 Wireshark 捕获的 TURN ChannelData 消息

有关 ICE 运行机制的详情，请参见第 9 章。本节将讨论 ICE 如何利用 STUN 和 TURN 协议。请注意，使用单个端口时，将采用多路复用技术传输 STUN、SRTP 和 DTLS，详见 [RFC5764] 的 5.1.2 节。

ICE 利用 STUN 的短期身份验证机制，对于每个会话，将随机生成用户名和密码，并在提议 / 应答交换中进行交换。双方均为会话贡献一半用户名（在 a=ice-ufrag SDP 属性中承载）和一个密码（在 a=ice-pwd SDP 属性中承载）。会话用户名是每个用户名片段的串联，所用密钥则是另一端的 ICE 代理发送的密码。MESSAGE-INTEGRITY 属性包含整个 STUN 消息的带密钥的消息鉴别码（带密钥的 HMAC 或散列函数）。

收集的候选地址按两种方式进行分类。所有非主机地址都有一个基址，该基址就是用于生成候选项的 IP 地址和端口。对于反射候选项，该基址是用于发送 STUN 检查的主机地址。对于中继候选项，该基址是用于转发中继通信的 IP 地址和端口。每个候选项都有一个基本字符串与之关联。基本字符串是一个唯一的字符串，用于将具有相同连接地址（IPv4 或 IPv6）、基本 IP 地址和传输协议（通常为 UDP），且使用相同 STUN 或 TURN 服务器派生的候选项归为一组。例如，如果两个候选项都包括 IPv4 地址、采用 UDP 传输协议并利用相同的 STUN 和 TURN 服务器，则具有相同的基本字符串。

ICE 通过定期发送数据包来提供长连接功能。有一项扩展内容（详见 11.3.10 节）可以允许在这个长连接上不断传输数据，要求客户端得到响应，否则就重新启动 ICE。

ICE 对等连接检查通过用于 RTP 媒体会话的相同端口发送。为帮助对协议进行多路传输，ICE 要求使用 STUN FINGERPRINT 属性。FINGERPRINT 属性包含一个循环冗余代码，此代码涵盖整个 STUN 消息。ICE 还要求使用 PRIORITY STUN 属性。PRIORITY 属性承载为候选项选定的优先级，并由另一端的 ICE 代理用于确定通过此连接检查生成的各个对

等端反射候选项的优先级。

表 10.7 显示了为 ICE 定义的新 STUN 属性。USE-CANDIDATE 属性由施控 ICE 代理用于选择要在连接检查中使用的候选项对。受控 ICE 代理将此属性包括在响应中以确认所选的候选项对。ICE-CONTROLLED 属性包括在受控 ICE 代理执行的所有连接检查中。ICE-CONTROLLING 属性包括在施控 ICE 代理执行的所有连接检查中。ICE 包括一种机制来选择哪个 ICE 代理是施控方，哪个代理是受控方。如果两个代理选择的角色相同，将可以通过这些属性发现此问题，届时将发送“487 Role Conflict”（487 角色冲突）响应并重新选择角色。

表 10.7 为 ICE 定义的 STUN 属性

属性	用途
USE-CANDIDATE	施控 ICE 代理指示要使用的候选项
ICE-CONTROLLED	由受控 ICE 代理包括的属性
ICE-CONTROLLING	由施控 ICE 代理包括的属性

为选择将用于传输媒体的特定候选项对，施控 ICE 代理会发送一个包含 USE-CANDIDATE 属性的连接检查请求。受控 ICE 代理将发送同样的报文响应连接检查。如果施控 ICE 代理希望选择使用最佳候选项对，则先完成所有连接检查，再选择成功通过检查且优先级最高的对。如果施控 ICE 代理希望使用第一个成功的候选项对，则会将 USE-CANDIDATE 属性包括在发送的每个连接检查中。受控 ICE 代理收到的第一个成功的连接检查将包含 USE-CANDIDATE 属性，施控 ICE 代理收到的第一个成功响应也将包含此属性。

WebRTC 使用的 ICE 将支持缓慢型 ICE 扩展内容（详见 8.5.1 节）。

通过将“Display Filter”（显示过滤器）设置为“stun”并查找表 10.7 中的 ICE 属性，可在 Wireshark 中找到 ICE 数据包，如图 10.14 所示。ICE 数据包将由用于媒体数据包的相同端口进行多路传输。

10.2.8 TLS 协议

传输层安全（Transport Layer Security，TLS）[RFC5246] 协议的前身称为安全套接字层（Secure Sockets Layer，SSL），这是 TCP 和应用程序之间的一个中介层，可提供加密和身份验证服务。加密服务通过对“传输中”的数据包进行加密来实现。身份验证服务则使用数字证书来实现。当今的安全 Web 浏览（HTTPS）仅使用 TLS 传输机制。WebRTC 可利用 TLS 来确保信令和用户界面的安全。此外，还有一个基于 UDP 运行的 TLS 版本，称为“数据报 TLS”（Datagram TLS，DTLS）（详见 10.2.10 节），以及一个用于为 SRTP 生成密钥的版本，称为“DTLSSRTP”[RFC5764]。

通过将“Display Filter”（显示过滤器）设置为“tcp”并查找 TLSv1 或 TLSv1.2 的协议，可在 Wireshark 中看到 TLS 握手信息，但无法对实际 TLS 数据包进行解码，因为 Wireshark 无法识别它们使用的加密密钥。

10.2.9 TCP 协议

传输控制协议（Transmission Control Protocol，TCP）[RFC793] 是 Internet 协议体系中的传输层协议，可提供可靠传输、拥塞控制和流量控制功能。TCP 用于传输 Web（HTTP）通信，但不适用于传输 RTP 等实时通信，因为用于实现可靠性的重新传输机制会导致过长的延迟。与 UDP 一样，TCP 使用端口的概念（一个 16 位整数）来分离数据流和协议。TCP 由浏览器所在的操作系统提供。

通过将“Display Filter”（显示过滤器）设置为“tcp”，Wireshark 可找出 TCP 数据包，其中将包括 HTTP 和 WebSocket，而且往往有信令通信。一般而言，媒体将使用 UDP 传输。

10.2.10 DTLS 协议

DTLS[RFC6347] 是基于 UDP 运行的 TLS 版本。与 TLS 一样，它也提供加密和身份验证属性。UDP 更易于穿透 NAT，因而更适合对等应用程序。

通过将“Display Filter”（显示过滤器）设置为“udp”并查找 DTLSv1.0 的协议，可在 Wireshark 中看到 DTLS 握手信息。在 Wireshark 中，实际 DTLS 数据包（例如数据通道消息）将仅显示为加密的 UDP 数据包。

10.2.11 UDP 协议

用户数据报协议（User Datagram Protocol，UDP）[RFC768] 是 Internet 协议体系中的传输层协议，可为其上的诸层提供不可靠的数据报服务。UDP 通常用于交换简短的小型数据包（例如 DNS 数据包）或传输 RTP 等实时媒体。UDP 提供了一种迅捷、高效的信息交换方式，但 UDP 用户必须处理可能丢失数据包的问题。此外，UDP 没有拥塞控制功能，因此用户必须能够敏感地识别数据包丢失和拥塞情形，以免造成 Internet 连接过载。与 TCP 一样，UDP 使用端口的概念（一个 16 位整数）来分离数据流和协议。

大多数 Internet 应用程序都采用可靠传输机制，例如传输控制协议（Transmission Control Protocol，TCP），以便自动重新传输丢失的数据包。浏览 Web、收发电子邮件以及流式传输音频和视频都使用可靠传输机制。对于收到的数据包，接收方都要进行确认，如果在特定的时间长度内没有确认，将触发重新传输数据包的操作，直至收到确认为止。实时通信无法利用这种可靠传输的机制，因为检测数据包是否丢失及接收重新发送的数据包都会带来巨大的延迟。如果加载网页时出现数据包丢失问题，将可能导致页面需要多用两到三秒才能加载完毕。在实时通信会话中，不能让正在进行的语音对话暂停一到两秒，亦不能让视频停放一会，以等待重新传输丢失的信息。实时通信系统必须能够在信息丢失时做出最佳处理。用于补偿数据包丢失或最大限度减少其影响的技术称为“数据包丢失隐藏”（Packet Loss Concealment，PLC）。

一般而言，Internet上极少丢失数据包，平均丢失率只有几百分之一。虽然数据包丢失情况并不经常出现，但往往会突然爆发，导致在短时间内丢失大量数据包。能否处理这种短时丢失事件，将显著影响通信系统带来的质量体验。高级编解码器，尤其是Opus音频编解码器（详见11.3.9节）经过专门设计，即使在数据包大量丢失时，亦能提供良好的用户体验。此外，来自媒体接收器的实时反馈还可在数据包拥塞期间提供减少带宽或分辨率的功能，从而提供更好的用户体验，并与其他Internet用户合理分享带宽。

UDP由浏览器所在的操作系统提供。

通过将“Display Filter”（显示过滤器）设置为“udp”，可在Wireshark中找出UDP数据包；该设置将查找媒体数据包（RTP）、控制数据包（RTCP）、数据通道数据包（加密DTLS），以及其他各种通信（包括DNS查询）。

10.2.12 SCTP协议

数据通道采用流控制传输协议（Stream Control Transport Protocol，SCTP）[RFC4960]构建。SCTP最初设计用于在IP网络上传输PSTN电话协议信令系统7（Signaling System #7，SS7）消息。它具备TCP所没有的实用功能，包括：

- 可靠或半可靠传输
- 无序传输数据包
- 在一个SCTP关联中传输多个流
- 能够发送消息

可靠或半可靠传输选项对Web开发人员极为有用。对于某些应用程序，可靠性是最重要的属性，例如在执行文件传输操作时。对于另外一些应用程序，半可靠传输足以满足要求，例如当传达的信息仅在一段有限的时间内有效时。在半可靠模式下，SCTP将尝试重新传输，但会在重新传输固定的次数或重新传输持续一定时间后放弃此操作。例如，实时位置信息（例如跟踪应用程序）和演讲人身份信息（例如在会议中）都非常适合采用半可靠传输模式。

无序传输可避免一个众所周知的TCP问题，称为“线头阻塞”（Head of Line Blocking）。在TCP中，流数据始终按顺序（即按照字节流顺序）传输至用户应用程序。如果某一段数据丢失，TCP栈向用户移交数据的过程将中止，直至收到丢失的数据段，即使在此期间收到了其他数据段，情况也不例外。在此类情况下，SCTP将会继续向用户移交数据，同时重新传输丢失的数据段。并非所有应用程序都能够利用此无序传输功能，但有一部分能够如此，而SCTP可提供此项功能。

SCTP提供在一个SCTP关联中传输多个流的概念。这些流将得到单独处理，甚至可以具有不同的可靠性属性。流与流之间不存在线头阻塞问题。流由流编号标识，可在一定程度上提供多路复用功能，而无需开通多个SCTP连接。

TCP面向流，而SCTP则面向消息。这意味着，使用SCTP的应用程序在收发各个消

息时无需管理自己的帧。

SCTP 还有其他一些未被 WebRTC 利用的属性，例如多宿主（Multi-homing），此属性允许在多个 IP 地址之间建立 SCTP 关联，从而实现回退和冗余。由于数据通道是 WebRTC 对等连接的一部分，因此它始终是两个浏览器之间或浏览器与其他设备之间的点对点连接。

与 TCP 一样，SCTP 内置有拥塞控制机制。在协议中，拥塞控制十分重要，有助于 Internet 的不同用户合理分享有限的带宽，并以缓解而非加重拥塞的方式应对拥塞情形。目前，IETF 的 RMCAT 工作组正在制定标准，旨在向 RTP 中加入拥塞避免和控制机制，并使其在设计上能够与 TCP 和 SCTP 拥塞控制实现有效配合。

基于 DTLS 传输 SCTP

与 TCP 或 UDP 一样，SCTP 是一种传输协议，从理论上讲可直接在 IP 上运行。出于多种因素，WebRTC 并未如此利用 SCTP。广泛使用的网络地址转换器（Network Address Translator，NAT）（详见第 9 章）让人们难以使用 TCP 或 UDP 之外的传输协议，因为路径中的每个 NAT 都必须能够识别该传输协议。由于 SCTP 及其他一些新传输协议（例如数据报拥塞控制协议（Datagram Congestion Control Protocol，DCCP））的部署十分有限，因此当今几乎没有 NAT 支持它们。此外，在 IP 之上运行的传输协议通常由计算机的操作系统运行和管理。因为 SCTP 由 OS 实施，所以无法按照 WebRTC 的需要来灵活配置和使用 SCTP。鉴于此，人们在浏览器（称为“用户空间”）中实施 SCTP。

在 WebRTC 中，SCTP 在数据报传输层协议（Datagram Transport Layer Protocol，DTLS）之上运行。DTLS 为 SCTP 提供隐私保护、完整性保护和身份验证。有关基于 DTLS 传输 SCTP 的信息，请参见 [draft-ietf-tsvwg-sctp-dtls-encaps]。先建立 DTLS 连接并完成身份验证，然后建立 SCTP 关联。有关 DTLS 和 SCTP 如何发现最大传输单元的详情，请参见 [draft-ietf-tsvwg-sctp-dtls-encaps]。

Wireshark 无法对数据通道数据包中使用的 SCTP 进行解码，因为它们通过加密的 DTLS 会话传输。要查找数据通道 SCTP 数据包，可先查找 RTP 数据包（详见 10.2.3 节），然后查找具有无效 RTP 版本 0 的数据包。

10.2.13 IP 协议

Internet 协议（Internet Protocol，IP）是支撑 Internet 的网络层协议。当前的 IP 版本 4（IP version 4，IPv4）[RFC791] 的唯一地址标识符（称为“IP 地址”）即将用尽。人们制定了 IP 版本 6（IP version 6，IPv6）[RFC2460] 来显著扩展地址空间，让 Internet 能够在 21 世纪继续迅猛发展。遗憾的是，虽然许多 Internet 主干网络、服务和网站当前都支持 IPv6，但对 IPv6 的支持和部署依然进展缓慢。而且，并非所有 Internet 服务提供商（Internet Service Provider，ISP）都支持这一版本。当涉及不同版本的 IP 时，协商媒体和数据传输的工作可通过 ICE 完成。双堆栈 WebRTC 浏览器完全可以基于 IPv4 运行 HTTP 并基于 IPv6 传输媒

体，亦可基于 IPv6 运行 HTTP 并基于 IPv4 传输媒体。

通过将“Display Filter”（显示过滤器）设置为“ip”，Wireshark 可对 IPv4 或 IPv6 数据包进行解码。

10.3 参考资料

[WIRESHARK] http://wireshark.org

[RFC2616] http://tools.ietf.org/html/rfc2616

[RFC6455] http://tools.ietf.org/html/rfc6455

[RFC3550] http://tools.ietf.org/html/rfc3550

[RFC3711] http://tools.ietf.org/html/rfc3711

[RFC4566] http://tools.ietf.org/html/rfc4566

[RFC5389] http://tools.ietf.org/html/rfc5389

[RFC5766] http://tools.ietf.org/html/rfc5766

[RFC5245] http://tools.ietf.org/html/rfc5245

[RFC5246] http://tools.ietf.org/html/rfc5246

[RFC5764] http://tools.ietf.org/html/rfc5764

[RFC793] http://tools.ietf.org/html/rfc793

[RFC6347] http://tools.ietf.org/html/rfc6347

[RFC768] http://tools.ietf.org/html/rfc768

[RFC4960] http://tools.ietf.org/html/rfc4960

[RMCAT-WG] http://tools.ietf.org/wg/rmcat

[draft-ietf-tsvwg-sctp-dtls-encaps] http://tools.ietf.org/html/draft-ietf-tsvwg-sctp-dtls-encaps

[RFC791] http://tools.ietf.org/html/rfc791

[RFC5764] http://tools.ietf.org/html/rfc5764

[RFC2460] http://tools.ietf.org/html/rfc2460

第 11 章 Chapter 11

IETF 文档

目前，有许多标准文档对 WebRTC 中使用的协议进行了定义。其中，有些是 Internet 草案，即这些 IETF 工作文档尚未最终发布，仍在不断调整和制定之中。还有一些已经以意见征求书（Request for Comments，RFC）的形式发布，成为 IETF 的标准文档。草案文档按照当前讨论和编辑各文档的工作组进行划分。有关 WebRTC 的其他 RFC，例如针对 RTP 和 SDP 的 RFC，将在下一章中介绍。

11.1 意见征求书

IETF RFC 按其 RFC 编号引用，不会随时间而更改。RFC 有多种来源，包括 RFC 编辑页面 [RFC-EDITOR]。本书中提供的 RFC 链接指向 IETF 网站上存储的一个版本；该版本设有超链接，旨在方便大家访问。

目前，尚没有 IETF WebRTC 文档发布为 RFC。

11.2 Internet 草案

IETF Internet 草案是 IETF 正在制定的文档。在最终发布为 RFC 之前，这些文档会经常变更。Internet 草案分为工作组文档或个人提交的文档。在发布为 RFC 之前，个人文档发生调整的幅度可能最大，就连文档名称也很有可能更改。有关 IETF 标准流程的详情，请参见附录 B。

11.3 RTCWEB 工作组 Internet 草案

表 11.1 中列出了 IETF RTCWEB 工作组制定的主要文档。以下几节内容将围绕这些文档展开讨论。

表 11.1 IETF RTCWEB 工作组文档

文档	标题	章节
概述	“概述：针对基于浏览器的应用程序的实时协议”	11.3.1 节
使用情形和要求	“Web 实时通信使用情形和要求”	11.3.2 节
RTP 的用法	“Web 实时通信（WebRTC）：媒体传输和 RTP 的用法”	11.3.3 节
安全体系结构	“RTCWEB 安全体系结构”	11.3.4 节
威胁模型	“RTCWeb 安全注意事项”	11.3.5 节
数据通道	“RTCWeb 数据通道”	11.3.6 节
数据协议	“WebRTC 数据通道建立协议”	11.3.7 节
JSEP	“JavaScript 会话建立协议”	11.3.8 节
音频	“WebRTC 音频编解码器和处理要求”	11.3.9 节
许可刷新	“使用 STUN 刷新许可”	11.3.10 节
传输	“RTCWEB 传输”	11.3.11 节

11.3.1 “概述：针对基于浏览器的应用程序的实时协议”[draft-ietf-rtcweb-overview]

此概述工作组 Internet 草案 [draft-ietf-rtcweb-overview] 简要说明了 WebRTC 使用的协议和体系结构。其最终目标是在标准 HTML5 浏览器中内置相应的功能来进行实时音频、视频和数据通信。除了内置用于对媒体流进行编码和解码的编解码器外，还将内置媒体处理功能，例如回音消除（以便进行免提或直接对讲操作，而无需耳机或即按即说按键）和数据包丢失隐藏。其关键目标是在两个浏览器之间建立多媒体会话，以便直接在二者（“对等端”）之间发送媒体数据包。这可以减少服务器的负载、处理工作量和带宽需求，尽可能缩短媒体路径中的延迟，并最大限度减少丢失数据包的情况。对于随网页加载而下载的 JavaScript Web 应用程序，将使用应用程序编程接口（Application Programming Interfaces，API）向其公开浏览器 RTC 功能。此文档从整体上说明了用于解决 WebRTC 问题的体系结构和方法。

11.3.2 “Web 实时通信使用情形和要求”[RFC7478]

RFC 7478 [RFC7478] 详细说明了 WebRTC 的要求和使用情形。这些要求包括：能够遍历网络地址转换（Network Address Translation，NAT），兼容 IPv4 和 IPv6 以及双协议栈浏览器，利用宽带和窄带 Internet 连接，以及能够处理拥塞和数据包丢失。使用情形包括包含多个源和流的音频和视频。该文档还描述了多方通信问题。应用领域包括传统电话呼叫、会聚式视频聊天、在游戏中进行对等信息交换，以及分布式音乐制作。此文档还讨论

了如何通过 SIP 和其他信令协议来结合使用公共交换电话网络（Public Switched Telephone Network，PSTN）以及现有的 IP 语音（Voice over IP，VoIP）和多媒体系统。

11.3.3 “Web 实时通信 (WebRTC)：媒体传输和 RTP 的用法”[draft-ietf-rtcweb-rtp-usage]

此工作组 Internet 草案 [draft-ietf-rtcweb-rtp-usage] 说明了如何在 WebRTC 中使用实时传输协议（Real-time Transport Protocol，RTP）。浏览器将拥有完整的 RTP 栈，该栈内置为 RTC 功能的一部分，如图 1.2 所示。此文档还说明了如何使用 RTP 控制协议（RTP Control Protocol，RTCP）来交换会话信息以及有关质量和拥塞情况的发送方和接收方报告。除了 10.2.3 节中介绍的核心 RTP 规范外，WebRTC 还对 RTP 实施了一系列扩增。其中，有些扩展内容十分常见，有些则较为罕见。此文档没有定义任何新的 RTP 扩展内容，但引用了其他定义了新 RTP 扩展内容的 RFC 和 Internet 草案。常规 RTP 与 WebRTC 使用的 RTP 之间最重要的区别在于，后者采用多路复用技术。通常，每个 RTP 媒体流均使用一个唯一的 UDP 端口号，而与给定 RTP 流关联的 RTCP 会话则使用另一个唯一的端口号。因此，在一个同时涉及音频、视频和关联 RTCP 会话的多媒体会话中，通常需要四个不同的 UDP 端口。然而，在 WebRTC 中，只需使用一个 UDP 端口即可，所有媒体、语音和视频及相应的 RTCP 会话都通过同一端口进行多路传输。这可以显著减少遍历 NAT 设备所需的工作量。有关通过一个端口对 RTP 和 RTCP 数据包进行多路传输的信息，请参见 12.1.5 节。

为了向后兼容非 WebRTC 端点（例如 SIP 或 Jingle 客户端），浏览器需要恢复为原先的设置，即在正常媒体协商中使用多个 UDP 端口。

此文档还引用了许多会议和标头扩展内容。

由于此文档中定义了在 WebRTC 中使用 RTP 和 RTCP 的要求，因此将发布为标准跟踪 RFC。

11.3.4 “RTCWEB 安全体系结构”[draft-ietf-rtcweb-security-arch]

此工作组 Internet 草案 [draft-ietf-rtcweb-security-arch] 说明了 WebRTC 的安全体系结构。在实时通信中，适用基本 Web 浏览安全模型。此模型最简单的形式就是用户必须信任其 Web 浏览器，用户依赖其浏览器来抵御自己可能访问的潜在恶意网站。在允许网站访问麦克风或摄像头之前，浏览器必须获得用户的许可。图 11.1 显示了一个真实示例，当 WebRTC 应用程序（Meetecho 协作工具 [MEETECHO]）向用户请求使用麦克风和摄像头的许可时，WebRTC 浏览器（Mac OS 上的 Google Chrome Canary）开始请求用户许可。许可请求位于 URL 栏下方。

此外，当网站使用麦克风或摄像头时，也必须向用户指示这一情况。图 11.2 显示了 Meetecho WebRTC 应用程序是如何体现此项要求的。

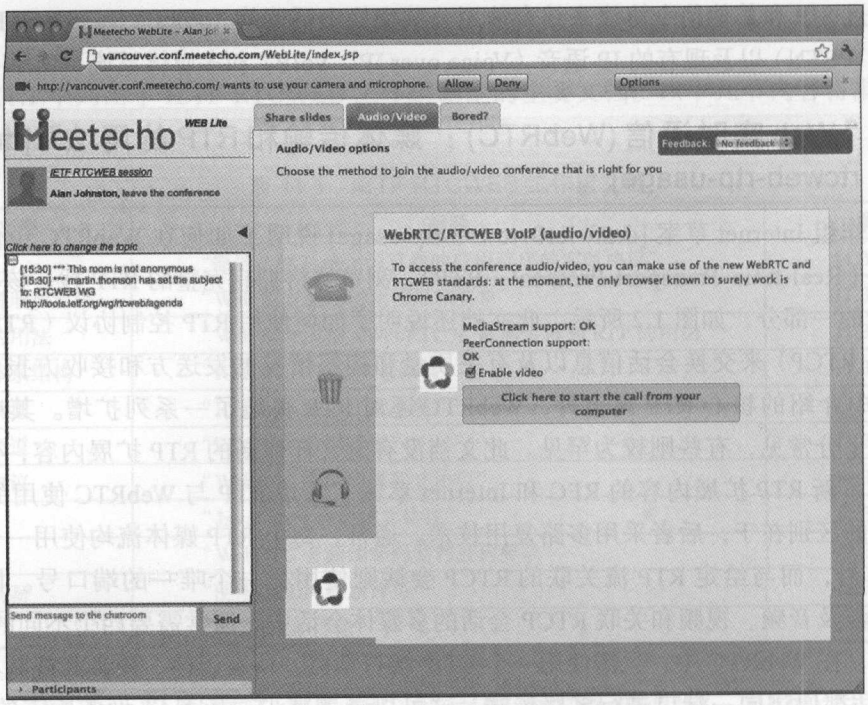


图 11.1 WebRTC 浏览器正在请求用户许可

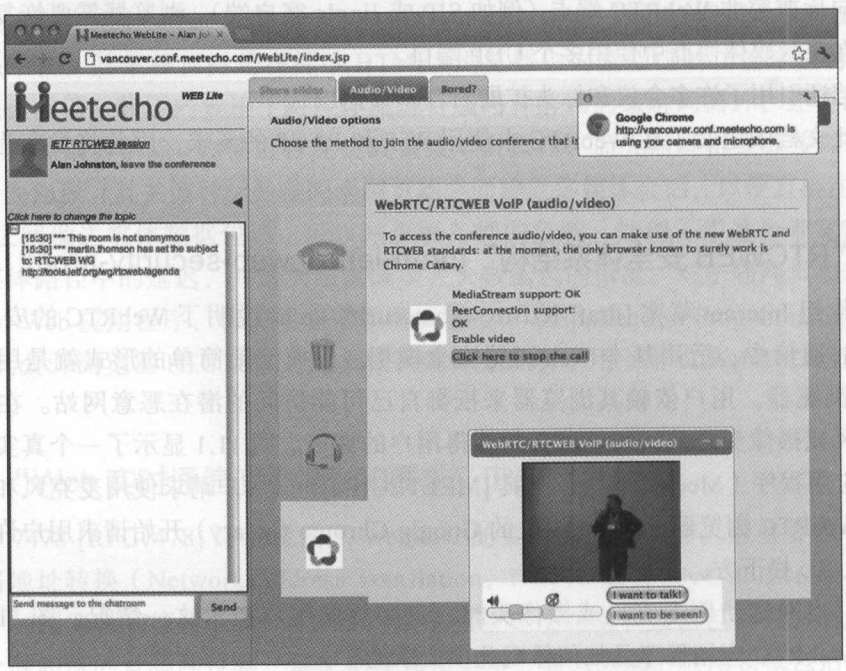


图 11.2 WebRTC 浏览器向用户指示正在使用麦克风和摄像头

此文档还讨论了如何使用 TLS（详见 10.2.8 节）、SRTP（详见 10.2.3 节）和 DTLS-SRTP [RFC5764] 等协议在 WebRTC 中提供安全防护。例如，文中讨论了如何使用 HTTPS 提供的安全功能，如图 11.3 所示。

有关 WebRTC 安全性及如何使用身份提供程序的详情，请参见第 13 章。

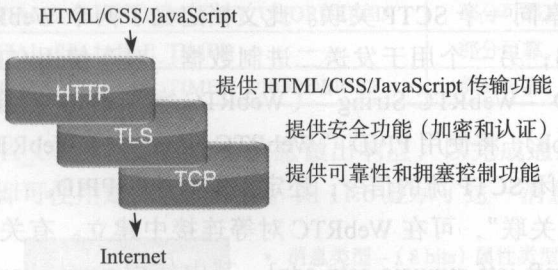


图 11.3 WebRTC 中的 HTTPS 安全层

11.3.5 “RTCWeb 安全注意事项”[draft-ietf-rtcweb-security]

此工作组 Internet 草案 [draft-ietf-rtcweb-security] 说明了 WebRTC 威胁模型，可用于为“RTCWEB 安全体系结构”文档提供论据。此威胁模型将用于评估协议规范中的安全机制。有关 WebRTC 安全性的详情，请参见第 13 章。

11.3.6 “RTCWeb 数据通道”[draft-ietf-rtcweb-data-channel]

此工作组 Internet 草案 [draft-ietf-rtcweb-data-channel] 讨论了在两个浏览器之间交换非 RTP、非媒体数据的要求和适用的协议。其提案是基于用户数据报协议（User Datagram Protocol，UDP）之上的数据报传输层安全协议（Datagram Transport Layer Security Protocol，DTLS）（详见 10.2.10 节）来使用流控制传输协议（Stream Control Transport Protocol，SCTP）（详见 10.2.12 节），如图 11.4 所示。此协议栈有些复杂，可提供 NAT 遍历、身份验证、加密和多流可靠传输功能。虽然 SCTP 是一种传输层协议，但由于存在 NAT，因而不能直接用于 Internet 协议（Internet Protocol，IP）之上。正确的做法是通过 UDP 为整个协议栈建立隧道，以免 NAT 丢弃数据包。在建立数据通道时，将使用 ICE 提供通信许可和打洞，以支持 NAT 和防火墙遍历。请注意，SCTP 将在浏览器本身中实现（因此称为“用户空间”），而不依赖操作系统（内核）实施方案。

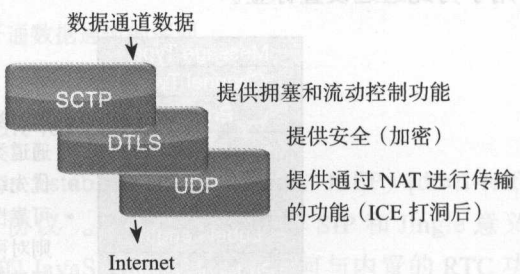


图 11.4 WebRTC 数据通道的协议层

11.3.7 “WebRTC 数据通道建立协议”[draft-ietfrtcweb-data-protocol]

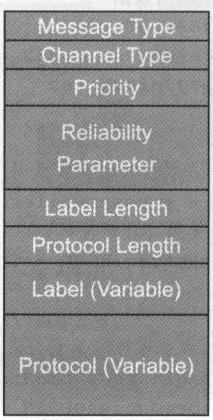
数据通道建立协议（Data Channel Establishment Protocol，DCEP）在 [draft-ietfrtcweb-data-protocol] 中定义。

每个 SCTP 消息都包含一个协议有效负载标识符（Protocol Payload Identifier，PPID），因而可使多个协议共享同一个 SCTP 关联。此文档定义了两个 WebRTC PPID：一个用于发送 JavaScript 字符串；另一个用于发送二进制数据。对于使用 UTF-8 编码的 JavaScript 字符串，将使用 PPID “WebRTC String”（WebRTC 字符串）。对于 JavaScript 二进制数据（ArrayBuffer 或 Blob），将使用 PPID “WebRTC Binary”（WebRTC 二进制数据）。对于 DCEP 中用于开通和关闭 SCTP 流的信令，还定义了额外的 PPID。

SCTP 连接称为“关联”，可在 WebRTC 对等连接中建立。有关用于建立数据通道的 SDP 信令，请参见 [draft-ietf-mmusic-sctp-sdp]，其中使用 m=application 媒体类型。SCTP 关联中的流为单向流，并以包含 16 位数字的 SCTP 流标识符进行标识。DCEP 定义了如何将两个单向 SCTP 流合并为一个双向 WebRTC 数据通道。DCEP 消息使用 SCTP PPID 或是“WebRTC 控件”或“WebRTC DCEP”。

DCEP 定义了两种消息类型：DATA_CHANNEL_OPEN 和 DATA_CHANNEL_ACK。为开通数据通道，需要选择一个 SCTP 流 ID。如果开通数据通道的一端是建立此 SCTP 关联时使用的 DTLS 连接的 DTLS 服务器（即执行被动开通），则选择奇数会话 ID。如果这一端是 DTLS 客户端（即执行主动开通），则从尚未使用的流 ID 中选择一个偶数会话 ID。这可以避免任何争用冲突（称为“双占用”）；所谓双占用，是指双方同时尝试使用同一会话 ID 编号打开通道。

图 11.5 显示了包含各字段的 DATA_CHANNEL_OPEN 消息。“Channel Type”（数据通道）字段指示所需的可靠性，它必须是下表 11.2 中显示的选项之一。“Priority”（优先级）字段指示此通道相对于通过该 SCTP 关联建立的其他通道的优先级。“Label”（标签）字段用于为此通道设置标签。



- 消息类型 –（8 bits）属性类型
- 通道类型 –（8 bits）通道可靠性类型
- 优先级（16 bits）此通道的相对优先级
- 可靠性参数 –（32 bits）如果通道为部分可靠，则对可靠性施加控制
- 标签长度 –（16 bits）标签字段的长度
- 协议长度 –（16 bits）协议字段的长度
- 标签 –（变量）通道的名称
- 协议 –（变量）通道的协议

图 11.5 DATA_CHANNEL_OPEN 消息

表 11.2 数据通道可靠性选项

通道类型	用法
DATA_CHANNEL_RELIABLE	可靠
DATA_CHANNEL_RELIABLE_UNORDERED	可靠、无序
DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT	部分可靠，重新传输次数固定
DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT_UNORDERED	部分可靠，无序，重新传输次数固定
DATA_CHANNEL_PARTIAL_RELIABLE_TIMED	部分可靠，定时重新传输
DATA_CHANNEL_PARTIAL_RELIABLE_TIMED_UNORDERED	部分可靠，无序，定时重新传输

另一端将以 DATA_CHANNEL_ACK 消息做出响应，以完成通道开通序列，但开通方不需要等待确认消息即可使用通道发送数据。图 11.6 显示了这一消息。

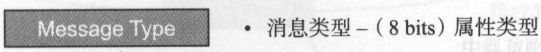


图 11.6 DATA_CHANNEL_ACK 消息

图 11.7 显示了建立数据通道的整个步骤序列。请注意，如果同时开通媒体会话，则 DTLS-SRTP 将使用 DTLS 连接为并行建立的 SRTP 媒体会话生成密钥。

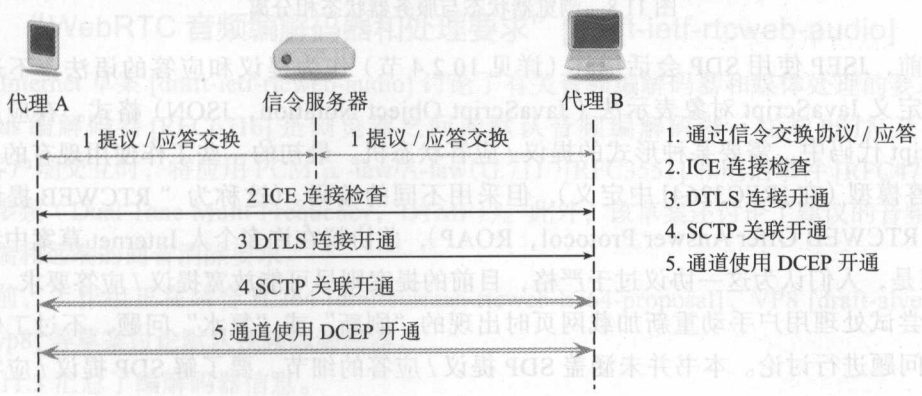


图 11.7 使用 DCEP 开通数据通道的序列

11.3.8 “JavaScript 会话建立协议”[draft-ietfrtcweb-jsep]

JavaScript 会话建立协议（JavaScript Session Establishment Protocol，JSEP）[draft-ietf-rtcwebjsep] 是为 WebRTC 开发的一种新“信令协议”。实际上，它并非 SIP 和 Jingle 意义上的那种信令协议，而是描述了浏览器中运行的 JavaScript 应用程序如何与内置的 RTC 功能进行交互。JSEP 定义了 JavaScript 如何获取有关浏览器功能的信息，包括支持的媒体类型和编解码器，即 SDP RTCSessionDescription 对象。它还描述了 JavaScript 如何管理浏览器之间的提议 / 应答媒体协商和浏览器中运行的 ICE 打洞进程。需要指出的是，JSEP 没有定义任何在线协议，即没有描述如何与 Web 服务器之间发送 SDP 对象。对此，可以使用第

4 章中介绍的任意方法。

熟悉信令协议的读者可能想知道“状态机”位于何处。WebRTC 中的状态机保留在浏览器中，但其运行完全处于 JavaScript 代码的控制之下。

API 用于从浏览器中获取候选项和功能信息。ICE 状态机在浏览器本地运行，并与 JSEP 状态机相分离。图 11.8 显示了浏览器状态和服务器状态相分离的设计。

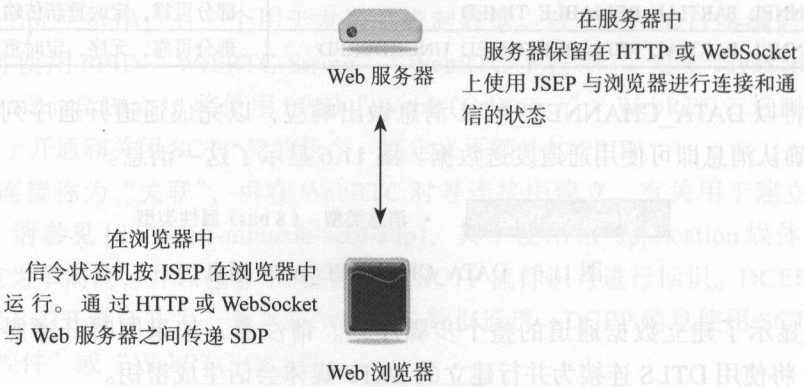


图 11.8　浏览器状态与服务器状态相分离

目前，JSEP 使用 SDP 会话描述（详见 10.2.4 节）作为提议和应答的语法，不过将来可能会定义 JavaScript 对象表示法（JavaScript Object Notation，JSON）格式。在应用程序 JavaScript 代码中，需要某种形式的提议 / 应答状态机。最初的一些工作使用现有的 SIP 提议 / 应答模型（在 [RFC3264] 中定义），但采用不同的编码。（这称为“ RTCWEB 提议应答协议”（RTCWEB Offer Answer Protocol，ROAP），此协议在许多个人 Internet 草案中都有描述。）但是，人们认为这一协议过于严格，目前的提案则尽可能放宽提议 / 应答要求。此外，此提案尝试处理用户手动重新加载网页时出现的“刷新”或“复水”问题，不过工作组仍在对此问题进行讨论。本书并未涵盖 SDP 提议 / 应答的细节。要了解 SDP 提议 / 应答，请参见 [SDP-OA]。

图 11.9 显示了 JSEP 状态机。

请考虑由某一端的浏览器发起新对等连接的情况。最初的状态为“闲置”，接下来，浏览器生成提议并将其发送给另一端的浏览器。此时的状态为“本地提议”。如果收到应答，将建立对等连接，此时会进入“活动”状态。此外，也可能收到临时应答。临时应答与应答完全相同，只是不应释放与提议关联的资源。对于临时应答，可能会收到多个，但只有收到应答才会建立对等连接并进入“活动”状态。对于收到提议的浏览器，将具有类似的状态机，只是浏览器通常不应发送临时应答，而应发送完整的应答。临时应答通常由网关发送给 SIP 等其他协议，用于处理分叉等特殊情形。一旦进入“活动”状态，任何发起新提议的浏览器都可以更改对等连接，并重复上述过程。

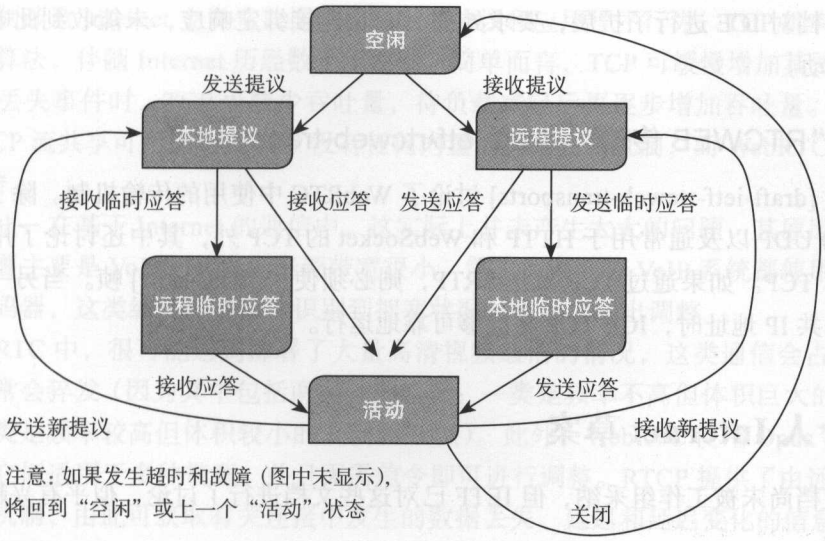


图 11.9 JSEP 状态机

11.3.9 “WebRTC 音频编解码器和处理要求”[draft-ietf-rtcweb-audio]

此 Internet 草案 [draft-ietf-rtcweb-audio] 讨论了有关音频编解码器和媒体处理的要求。例如，Opus 编解码器 [RFC6716] 是浏览器之间的默认音频编解码器，在与 PSTN 及 SIP 和 Jingle 客户端交互时，将应用 PCM μ-law/A-law(G.711)[RFC3551] 和电话事件 [RFC4733](针对双音多频（Dual Tone Multi Frequency，DTMF））。此外，该草案还讨论了建议的音频级别，以及近端和远端的回音消除要求。

目前，工作组正在结合 H.264 [draftburman-rtcweb-h264-proposal]、VP8 [draft-alvestrand-rtcweb-vp8] 等草案讨论默认视频编解码器。

表 11.3 汇总了编解码器信息。

表 11.3 WebRTC 编解码器摘要

编解码器	用法	规范
Opus	适用于语音和音乐的窄带到宽带 Internet 音频编解码器	RFC 6716
G.711	适用于 PSTN 交互的 PCM 音频编码，向后兼容 VoIP 系统	RFC 3551
电话事件	传输 DTMF 提示音	RFC 4733
H.264	提议的视频编解码器，需要获得许可	RFC 6184
VP8	提议的开源视频编解码器	RFC 6386

11.3.10 “使用 STUN 刷新许可”[draft-ietf-rtcweb-stunconsent-freshness]

此草案 [draft-ietf-rtcweb-stun-consent-freshness] 讨论了一些重要的潜在变更事项，涉及浏览器如何确定多媒体会话是否仍处于活动状态，以及另一方是否希望继续收到协商的

媒体。此文档对 ICE 进行了扩展，要求处理 ICE 长连接绑定响应，未能收到此响应将导致 ICE 重新启动。

11.3.11 “RTCWEB 传输”[draft-ietf-rtcweb-transports]

此草案 [draft-ietf-rtcweb-transports] 讨论了 WebRTC 中使用的传输机制。除了通常用于 RTP 媒体的 UDP 以及通常用于 HTTP 和 WebSocket 的 TCP 外，其中还讨论了用于媒体的 TCP 和 ICE TCP。如果通过 TCP 发送 SRTP，则必须使用 [RFC4571] 帧。当另一端位于云中或拥有公共 IP 地址时，ICE TCP 将能够可靠地运行。

11.4 个人 Internet 草案

这些文档尚未被工作组采纳，但 IETF 已对这些文档进行了讨论，似乎有兴趣对它们提供一些支持。

11.4.1 “用于 RTCWeb 媒体约束的 IANA 注册表”[draftburnett-rtcweb-constraints-registry]

此个人 Internet 草案 [draft-burnett-rtcweb-constraints-registry] 定义了一个 Internet 编号分配机构（Internet Assigned Numbers Authority，IANA）注册表，用于跟踪 RTCPeerConnection 和 getUserMedia 接口中使用的所有媒体约束和功能。

11.4.2 “关于 NAT、防火墙和 HTTP 代理的 RTCWEB 注意事项”[draft-hutton-rtcweb-nat-firewall-considerations]

此个人 Internet 草案 [draft-hutton-rtcweb-nat-firewall-considerations] 讨论了适用于 WebRTC 的 NAT 和防火墙注意事项。

11.4.3 “适用于 RTCWeb QoS 的 DSCP 和其他数据包标记”[draftdhesikan-tsvwg-rtcweb-qos]

此个人 Internet 草案 [draft-dhesikan-tsvwg-rtcweb-qos] 讨论了适用于 WebRTC 服务质量（Quality of Service，QoS）的区分服务代码点（DiffServe Code Point，DSCP）。它取代了 RTCWEB 工作组中早先名为 draft-ietf-rtcweb-qos 的草案（没有再进一步编写）。

11.4.4 “适用于万维网实时通信的 Google 拥塞控制”[draft-alvestrand-rmcat-congestion]

此个人 Internet 草案 [draft-alvestrand-rmcat-congestion] 描述了 Google 当前在其提交给 RTP 媒体拥塞规避技术工作组的 WebRTC 开源项目中实施的拥塞控制。

拥塞控制涉及 Internet 上发生数据包丢失时的协议和应用程序行为。TCP 拥有非常先进的拥塞控制算法，伴随 Internet 历经数十年发展。简单而言，TCP 可缓慢增加其吞吐量。当发生数据包丢失事件时，TCP 可减少吞吐量，待负载减轻后再逐步增加吞吐量。这可使多个不同的 TCP 流共享可用带宽。UDP 没有任何内置的拥塞控制机制，而 WebRTC 媒体流利用 UDP 传输。

迄今为止，在基于 Internet 的通信中，这实际上并未产生太大的问题，其原因有二。第一，这些部署主要是 VoIP，其中的音频带宽很小。第二，大部分 VoIP 系统都使用固定速率的电话编解码器，这类编解码器即使识别到拥塞状况，也无法做出调整。

在 WebRTC 中，很可能遇到部署了大量高清视频通信的情况，这类通信会占用大量带宽，并且经常会猝发（因为其中包括两类帧的组合：一类是频率不高但体积巨大的 I 帧或关键帧；另一类是频率较高但体积较小的 P 帧和 B 帧）。此外，WebRTC 使用 Opus 等 Internet 编解码器，它们适用于多种带宽，并且无需信令即可进行调整。RTCP 提供了由远程对等端进行反馈的机制，由此可获取有关连接中发生的数据丢失、延迟和延迟变化的信息。

IETF 开展了大量工作来开发适合基于 UDP 的 RTP 流的新拥塞控制算法。这些算法将利用从 RTCP 反馈中获取的网络统计数据，以便估计网络状况并检测拥塞，防止数据包丢失。这些信息将用于对 SRTP 媒体流和数据通道流使用的带宽进行限速。

此 Internet 草案中介绍的方法使用基于临时最大媒体流比特率请求（Temporary Maximum Media Stream Bit Rate Request，TMMBR）的 RTCP 反馈；其中，TMMBR 读作“timber”，详见 9.1.11 节。此方法使用信号处理技术来检测拥塞，防止数据包丢失。它还使用 TCP 友好速率控制（TCP Friendly Rate Control，TFRC），详见 12.1.10 节。另一个草案 [draft-alvestrand-rmcat-remb] 描述了用于传输接收器估计最大比特率（Receiver Estimated Max Bitrate，REMB）的 RTCP 消息。

今后，IETF 将尝试建立适用于 WebRTC 浏览器的标准 RTP 拥塞控制算法。2012 年 7 月，Internet 体系结构委员会（Internet Architecture Board，IAB）举办了一场主题为“交互式实时通信拥塞控制”的研讨会。有关该研讨会的信息，包括会上讨论的文章的链接，均可在 IAB 网站 [IAB-CCIRTC] 上找到。他们成立了 RMCAT 工作组来开发用于 RTP 媒体的拥塞控制机制。

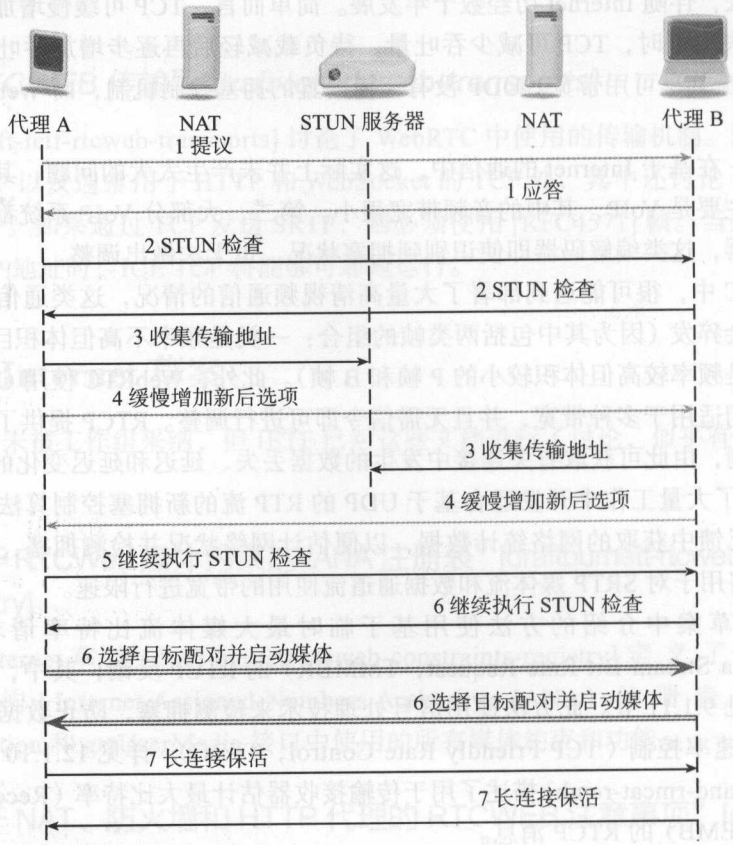
11.5 其他工作组的 RTCWEB 文档

当前为 WebRTC 开发的部分协议由 RTCWEB 之外的工作组制定，如 B.3 节所述。本节列出了这些 Internet 草案。

11.5.1 “缓慢型 ICE：逐步为交互式连接建立协议增加候选项的配置”[draft-ietf-mmusic-trickle-ice]

缓慢型 ICE [draft-ietf-mmusic-trickle-ice] 是对 ICE 的优化，旨在缩短完成 ICE 所需的

时间。图 11.10 中显示了缓慢型 ICE。



1）通过信令通道交换主机候选项
2）开始连接检查
3）收集更多传输地址
4）缓慢增加新候选项
5）继续执行连接检查
6）选择选定的对并启动媒体
7）发送长连接保活请求

图 11.10　缓慢型 ICE

缓慢型 ICE 并不像图 9.1 中所示的那样只有等到所有候选项都收集完毕才开始 ICE，而是只要有至少一个候选项可用，便开始进行 ICE 处理。这确保能够以最快的速度收集主机候选项，因为它不需要发送任何 STUN 或 TURN 消息。主机候选项可在提议 / 应答交换中交换，一旦交换完毕，即可开始 ICE 处理。随后，可并行添加或“缓慢加入”来自 STUN 服务器的服务器反射地址。如果主机或反射候选项对似乎都不能正常工作，还可以获取中继候选项并缓慢加入处理过程。这可以带来额外的优势，即只有在可能需要中继候选项时才预留它们。

11.5.2 “利用会话描述协议端口号进行多路协商”[draft-ietf-mmusic-sdp-bundle-negotiation]

此MMUSIC工作组Internet草案[draft-ietf-mmusic-sdpbundle-negotiation]由MMUSIC工作组讨论制定，其中定义了一种新的SDP分组框架扩展项，称为BUNDLE（a=group:BUNDLE）。有关SDP分组的定义，请参见[RFC5888]。通过这种分组，划分在一起的m=媒体线路可共享同一端口号。WebRTC将利用此分组在同一端口中以多路复用技术传输多种媒体的信号。目前，此草案仍在开发之中。

为使用BUNDLE，对于每个媒体线路，SDP提议或应答都将包含一个媒体分组ID（a=mid）。此外，还将有一个会话级别的分组属性（a=group:BUNDLE），用于列出捆绑在一起的每个媒体线路的媒体分组ID。捆绑在一起的媒体线路将使用同一传输地址来传输由这些媒体线路定义的所有RTP、RTCP和数据。BUNDLE可包含在任何SDP提议中，亦可在提议含有BUNDLE的情况下包含在相应的SDP应答中。

11.5.3 “会话描述协议中的跨流标识”[draft-ietf-mmusic-msid]

此MMUSIC工作组Internet草案文档[draft-ietfmmusic-msid]定义了a=msid属性扩展项，用于在SDP提议和应答中表示JavaScript MediaStream标识符字符串。此草案还定义了a=msid-semantic属性扩展项。其中还为WebRTC媒体流的WebRTC WMS定义了语义令牌。此扩展项用于将MediaStream关联至SSRC。正如WebRTC草案中讨论的那样，每个MediaStream由多个MediaStreamTrack构成。12.2.1节的SDP示例中显示这样一个例子。在最终敲定此扩展内容之前，还需要完成更多工作。请注意，此草案的-00版本建议由a=ssrc属性承载msid信息。Chrome浏览器版本32实施的似乎就是此版草案。

11.5.4 “RTP会话中的多种媒体类型”[draft-ietf-avtcore-multi-media-rtp-session]

此Internet草案[draft-ietf-avtcore-multi-media-rtp-session]说明了如何对RTP进行多路传输、RFC 3550中针对多路复用讨论的问题，以及如何实现多路复用。目前，此草案正由AVTCORE工作组讨论。

11.5.5 “多媒体拥塞控制：用于单播RTP会话的断路器”[draft-ietf-avtcore-rtp-circuit-breakers]

此Internet草案[draft-ietf-avtcore-rtp-circuit-breakers]讨论了RTP发送方在哪些情况下应当停止发送数据，以免加剧拥塞状况。这种保护类似于在电路中的电流过高时中断电流的断路器。WebRTC最终也会实施相应的拥塞控制算法，用于在形成拥塞之前减少流量。

11.5.6 “在一个RTP会话中支持多个时钟速率”[draftietf-avtext-multiple-clock-rates]

此AVTEXT工作组Internet草案[draft-ietf-avtext-multipleclock-rates]就媒体会话中的SSRC时钟速率变化问题提供指导。例如，当在Opus和PCM编解码器之间进行切换时，由于二者使用不同的时钟速率，因此WebRTC中就会出现时钟速率改变的情况。

11.5.7 “会话描述协议中基于流控制传输协议(SCTP)的媒体传输”[draft-ietf-mmusic-sctp-sdp]

数据通道可通过交换SDP提议/应答来建立，如[draft-ietf-mmusic-sctp-sdp]中所述。建立数据通道时，需要使用m=application SDP媒体线路以及一个必选的a=sctpmap属性。例如：

```
m=application 54111 DTLS/SCTP 50002
a=sctpmap:50002 webrtc-datachannel 1
a=setup:actpass
a=connection:new
```

应用程序媒体线路包含UDP端口号（54111）和传输机制。请注意，在本例中，对于基于UDP上的DTLS运行的SCTP（通常写作SCTP/DTLS），协议令牌实际上是DTLS/SCTP，这看起来像是DTLS反过来位于SCTP之上。接下来是一种格式（在SDP语法中为<fmt>），其中包含SCTP端口号（本例中为50002）。再后面是a=sctpmap属性，它标识对该SCTP端口号应用DTLS/SCTP传输的协议（本例中为webrtcdatachannel）。随后是一个可选的最大流数，其值为1。如果此值不存在，则最大流数为16。由于DTLS是一种面向连接的协议，因此除了连接的new/existing状态外，还需要协商主动方/被动方角色。a=setup:actpass属性指示这一端既可作为主动方（DTLS客户端），亦可作为被动方（DTLS服务器），具体由另一端进行选择。a=connection:new属性指示将建立新的DTLS连接。

此草案还定义了用于DTLS/SCTP（同样，该定义中使用令牌SCTP/DTLS，似乎SCTP反过来位于DTLS之上）的SDP以及专门用于SCTP的SDP，不过WebRTC数据通道中并不使用它们。

请注意，一个SCTP/DTLS关联可用于多个协议。例如，如果要同时使用WebRTC数据通道和二进制发言权控制协议（Binary Floor Control Protocol，BFCP）会话，则SDP将更改为：

```
m=application 54111 DTLS/SCTP 50002 50006
a=sctpmap:50002 webrtc-datachannel 1
a=sctpmap:50006 bfcp 2
```

11.5.8 “会话描述协议中的媒体源选择机制”[draft-lennox-mmusic-sdp-source-selection]

此个人Internet草案[draft-lennox-mmusic-sdp-source-selection]对特定于源的媒体属性

进行了扩展，允许按源选择媒体流。a=remote-ssrc 属性用于选择一个特定的流并设置一个特定的属性。此规范还定义了一系列可使用此机制设置的媒体属性，包括 recv、framerate、imageattr 和 priority。recv 属性用于启用（recv:on）或禁用（recv:off）源。如果不存在 recv 属性，则采用 recv:on。framerate 属性用于请求视频流的特定帧速率。imageattr 属性用于设置媒体源的图像分辨率。priority 属性用于设置媒体源之间的相对优先级。如果带宽或其他限制导致无法收到请求的所有源，则使用优先级来确定应忽略或缩减哪些源。

此规范还定义了两个新的参数用于特定于源的媒体属性：information 和 sending。information 属性用来以类似于 SDP 中 i= 字段的方式，提供可供人理解的有关媒体源的文本。sending 属性指示特定源的发送状态，其值为 on 或 off。如果接收方禁用了某个源，或者发送方可能不再希望发送该源，则该源的状态就可能为 off。

在提议 / 应答 SDP 交换中，各方都应列出所有可用的源。请注意，还可以通过其他机制来发现源，例如通过 RTP 或其他会议通知收到 SSRC。

11.5.9 TRAM 工作组对 STUN 和 TURN 进行的扩展

目前，IETF 中新成立的 TURN 修订和革新（TURN Revised and Modernized，TRAM）工作组 [TRAM-WG] 正在对 STUN 和 TURN 制定扩展内容。虽然此工作组的名称中带有 TURN，但当前制定的许多扩展内容实际上都是 STUN 扩展项。本节列出了当前备受讨论的若干个人 Internet 草案。

TRAM 工作组计划更新 STUN 和 TURN 规范。其中，一个重要的方面是增加对 STUN 和 TURN 的支持，以实现比当前支持的 SHA1 散列更强大的加密算法。目前还没有 Internet 草案定义这种新的加密算法。

一种扩展是定义适用于 STUN 和 TURN 的传输，详见 [draft-petithuguenin-tram-stun-dtls]。DTLS 传输既可提供 TLS 隐私保护和身份验证的好处，又能避免必须通过 TCP 传输媒体的问题。通过使用 TLS 或 DTLS，浏览器还可以检查 TURN 服务器提供的证书，从而对 TURN 服务器进行身份验证。

另一个 STUN 扩展提议是在 STUN 和 TURN 请求中增加 ORIGIN 属性，详见 [draft-johnston-tram-stun-origin]。在各种浏览器解决方案中，ORIGIN 就是创建对等连接的页面的 HTTP 来源（包括 HTTP 或 HTTPS、域名和端口号）。来源信息为 STUN 和 TURN 服务器提供了额外的信息来处理请求、对请求进行验证身份，以及记录日志或进行调试。例如，可以配置企业 STUN 服务器，使之仅响应从与企业关联的来源发来的 STUN 绑定请求。对于支持多个域的 TURN 服务器（例如多租户服务器），可通过 ORIGIN 提示来确定要将哪个 REALM 包括在身份验证质询中。有关此问题以及现已发现的其他 TURN 身份验证问题，请参见 [draft-reddy-behave-turnauth]。如果不使用此方法或某种其他方法，TURN 服务器将需要为每个域提供唯一的 URI（不同的 IP 地址或端口）才能选择适当的 REALM。

另一个 TURN 扩展提议是在 TURN 中增加对 OAuth [RFC6749] 身份验证的支持，详见

[draft-reddy-tram-turn-third-party-authz]。该规范提议增加两个新的属性：THIRD-PARTY-AUTHORIZATION 和 ACCESSTOKEN。

[draft-patiltram-turn-serv-disc] 建议了两种发现 TURN 服务器的新方法。一种方法是使用动态主机配置协议（Dynamic Host Configuration Protocol，DHCP）和命名机构指针记录（Naming Authority Pointer Record，NAPTR）类型的域名服务（Domain Name Service，DNS）记录。另一种方法提议对 TURN 使用标准任意广播 [RFC4786] 地址。任意广播机制可将数据包路由到在拓扑结构中距离发送方最近的一组主机之一。

另一个草案提议对 STUN 进行新的扩展，以便在应用程序和网络之间启用有关服务质量（Quality of Service，QoS）的通信，详见 [draft-martinsen-tram-discuss]。此方法称为“使用 STUN 信令的差别优先级和状态代码点”（Differentiated prIorities and Status Code-points Using Stun Signalling，DISCUSS），它提议设置一系列新的属性，例如 STREAM-TYPE、BANDWIDTH-USAGE、STREAM-PRIORITY 和 NETWORK-STATUS。

还有一个草案提议为 TURN 设置 BANDWIDTH 属性，详见 [draftthomson-tram-turn-bandwidth]。通过此属性，TURN 客户端可向 TURN 服务器指示其希望在通过分配的中继地址收发数据时使用的最大带宽，TURN 服务器亦可向客户端指示应用限速之前允许的最大带宽。这有助于在用户之间公平共享 TURN 带宽，并可使 TURN 服务器与客户端共享策略。

请注意，目前所有这些 Internet 草案都是个人提交的文档。未来，其中一些草案可能会被采纳为 TRAM 工作组文档，届时草案名称将发生更改。要查找 TRAM 文档的最新版本，请使用链接 [TRAM-WG]。

11.6 参考资料

[RFC-EDITOR] http://www.rfc-editor.org

[draft-ietf-rtcweb-overview] http://tools.ietf.org/html/draft-ietf-rtcweb-overview

[RFC7478] http://tools.ietf.org/html/rfc7478

[draft-ietf-rtcweb-rtp-usage] http://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage

[draft-ietf-rtcweb-security-arch] http://tools.ietf.org/html/draft-ietf-rtcweb-security-arch

[MEETECHO] http://www.meetecho.com

[RFC5764] http://tools.ietf.org/html/rfc5764

[draft-ietf-rtcweb-security] http://tools.ietf.org/html/draft-ietf-rtcweb-security

[draft-ietf-rtcweb-data-channel] http://tools.ietf.org/html/draft-ietf-rtcweb-data-channel

[draft-ietf-rtcweb-data-protocol] http://tools.ietf.org/html/draft-ietf-rtcweb-data-protocol

[draft-ietf-mmusic-sctp-sdp] http://tools.ietf.org/html/draft-ietf-mmusic- sctp-sdp

[draft-ietf-rtcweb-jsep] http://tools.ietf.org/html/draft-ietf-rtcweb-jsep

[RFC3264] http://tools.ietf.org/html/rfc3264

[SDP-OA] “SIP:Understanding the Session Initiation Protocol”（SIP：了解会话初始协议）的第 13 章，第 3 版。

[draft-ietf-rtcweb-audio] http://tools.ietf.org/html/draft-ietf-rtcweb-audio

[RFC6716] http://tools.ietf.org/html/rfc6716

[RFC3551] http://tools.ietf.org/html/rfc3551

[RFC4733] http://tools.ietf.org/html/rfc4733

[draft-burman-rtcweb-h264-proposal] http://tools.ietf.org/html/draft-burman-rtcweb-h264-proposal

[draft-alvestrand-rtcweb-vp8] http://tools.ietf.org/html/draft-alvestrand-rtcweb-vp8

[draft-ietf-rtcweb-stun-consent-freshness] http://tools.ietf.org/html/draft-ietf-rtcweb-stun-consent-freshness

[draft-ietf-rtcweb-transports] http://tools.ietf.org/html/draft-ietf-rtcweb-transports

[RFC4571] http://tools.ietf.org/html/rfc4571

[draft-burnett-rtcweb-constraints-registry] http://tools.ietf.org/html/draft-burnett-rtcweb-constraints-registry

[draft-hutton-rtcweb-nat-firewall-considerations]

http://tools.ietf.org/html/draft-hutton-rtcweb-nat-firewall-considerations

[draft-dhesikan-tsvwg-rtcweb-qos] http://tools.ietf.org/html/draft-dhesikan-tsvwg-rtcweb-qos

[draft-alvestrand-rmcat-congestion] http://tools.ietf.org/html/draft-alvestrand-rmcat-congestion

[draft-alvestrand-rmcat-remb] http://tools.ietf.org/html/draft-alvestrand-rmcat-remb

[IAB-CCIRTC] http://www.iab.org/activities/workshops/cc-workshop/

[draft-ietf-mmusic-trickle-ice] http://tools.ietf.org/html/draft-ietf-mmusic-trickle-ice

[draft-ietf-mmusic-sdp-bundle-negotiation] http://tools.ietf.org/html/draft-ietf-mmusic-sdp-bundle-negotiation

[RFC5888] http://tools.ietf.org/html/rfc5888

[draft-ietf-mmusic-msid] http://tools.ietf.org/html/draft-ietf-mmusic-msid

[draft-ietf-avtcore-multi-media-rtp-session] http://tools.ietf.org/html/draft-ietf-avtcore-multi-media-rtp-session

[draft-ietf-avtcore-rtp-circuit-breakers] http://tools.ietf.org/html/draft-ietf-avtcore-rtp-circuit-breakers

[draft-ietf-avtext-multiple-clock-rates] http://tools.ietf.org/html/draft-ietf-avtext-multiple-clock-rates

[draft-ietf-mmusic-sctp-sdp] http://tools.ietf.org/html/draft-ietf-mmusic-sctp-sdp

[draft-lennox-mmusic-sdp-source-selection] http://tools.ietf.org/html/draft-lennox-mmusic-sdp-source-selection

[TRAM-WG] http://tools.ietf.org/wg/tram/

[draft-petithuguenin-tram-stun-dtls] http://tools.ietf.org/html/draft-petithuguenin-tram-stun-dtls

[draft-johnston-tram-stun-origin] http://tools.ietf.org/html/draft-johnston-tram-stun-origin

[draft-reddy-behave-turn-auth] http://tools.ietf.org/html/draft-reddy-behave-turn-auth

[RFC6749] http://tools.ietf.org/html/rfc6749

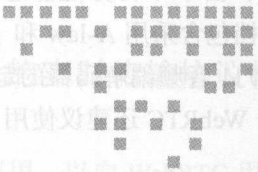
[draft-reddy-tram-turn-third-party-authz] http://tools.ietf.org/html/draft-reddy-tram-turn-third-party-authz

[draft-patil-tram-turn-serv-disc] http://tools.ietf.org/html/draft-patil-tram-turn-serv-disc

[RFC4786] http://tools.ietf.org/html/rfc4786

[draft-martinsen-tram-discuss] http://tools.ietf.org/html/draft-martinsen-tram-discuss

[draft-thomson-tram-turn-bandwidth] http://tools.ietf.org/html/draft-thomson-tram-turn-bandwidth



第 12 章 Chapter 12

与 IETF 相关的 RFC 文档

WebRTC 采用了纳入意见征求书（Request for Comments，RFC）的多项 IETF 标准和协议。这些 RFC 并非专为 WebRTC 开发或使用。本章基于相关协议列出了这些 RFC。

12.1 实时传输协议

12.1.1 “RTP：用于实时应用程序的传输协议”[RFC3550]

RFC 3550 [RFC3550] 定义了实时传输协议（Real-time Transport Protocol，RTP）的版本 2 和 RTP 控制协议（RTP Control Protocol，RTCP）。RTP 包含面向位的标头字段，用于承载有效负载类型（编解码器）、时间戳、序列号和同步源（Synchronization Source，SSRC）等信息。RTCP 消息包含发送方报告（Sender Report，SR）、接收方报告（Receiver Reports，RR）和源描述（Source Description，SDES）。（请注意，术语 SDES 是 SDP 安全描述（SDP Security Descriptions，在 [RFC4568] 中定义）的非正式名称——这两个概念并不相关。）SDES 消息承载规范名称（Canonical Name，CNAME），此名称用于在 RTP 会话中标识用户。利用参与源（Contributing Source，CSRC）字段，RTP 混合器可以提供数据包中所含媒体的发送方的相关信息。

12.1.2 “用于音频和视频会议的 RTP 配置文件”[RFC3551]

RFC 3551 [RFC3551] 定义了基本 RTP 音频和视频配置文件（Audio and Video Profile，AVP）。其中定义了多种常用音频和视频编解码器的格式，以及静态有效负载类型（值

0-95）。请注意，由于不再分配静态有效负载类型，因此必须改用动态有效负载类型（值 *96-127*）。*此文档包含采用 A-law 和 μ-law 压扩（压缩和扩展的混成词，可实现音频级别压缩）*的 PCM G.711 音频编解码器的定义。此文档需要进行更新才能供 WebRTC 使用，因为除 G.711 以外，WebRTC 还建议使用 DVI4 作为音频编解码器。

12.1.3 “安全实时传输协议”[RFC3711]

RFC 3711 [RFC3711] 定义了用于 RTP 的安全音频视频配置文件（Secure Audio Video Profile，SAVP）。其中提供了安全 RTP（Secure RTP，SRTP）和安全 RTCP（Secure RTCP，SRTCP）的用法。SRTP 使用对称密钥来加密和解密媒体与控制消息，由此向 RTP 提供加密和身份验证。SRTP 使用高级加密标准的计数器模式（Advanced Encryption Standard in Counter Mode，AES-CM）。SRTP 使用 128 位密钥，但在 [RFC6188] 中，它已得到扩展，可使用 192 和 256 位密钥。SRTP 需要一种密钥管理协议来确保发送方和接收方拥有相同的对称密钥。WebRTC 仅使用 DTLS-SRTP [RFC5764] 来管理密钥。SRTP 会生成加密的密钥流，此密钥流随后与媒体或控制数据包一起接受异或运算，以实现加密。通过此方式，可并行生成密钥流与媒体或控制数据包，从而将增加的延迟降至最低。此加密仅应用于 RTP 正文，RTP 标头和 RTP 标头扩展项不受影响。身份验证由一个新增的身份验证标记提供，此标记的长度为 0 ～ 10 个八进制位。每个媒体流都需要具有唯一的会话密钥。如果浏览器在会话中发送两个视频流和两个音频流，则将使用 4 个唯一的会话密钥来对其进行加密。

12.1.4 “用于基于 RTCP 的反馈且经过扩展的安全 RTP 配置文件 (RTP/SAVPF)”[RFC5124]

WebRTC 使用 SAVPF，其全称为“用于基于 RTCP 的反馈且经过扩展的安全 RTP 配置文件”[RFC5124]。比较而言，现今基于 Internet 的大多数通信 VoIP 和视频系统都使用常规 AVP 或 SAVP。SAVPF 配置文件既拥有 SAVP [RFC3711] 中 SRTP 的安全性，又具备 AVPF 配置文件 [RFC4585] 的及时反馈功能。[RFC3551] 中定义的基本 AVP 配置文件包括 RTCP 反馈消息，并拥有一套控制机制，可确保不将过多带宽用于这些控制消息，即使在大型会议中也是如此。此机制的一个缺点在于，接收方无法始终在最有利于发送方的时间发送反馈消息。为改进此反馈的及时性，AVPF 引入了早期 RTCP 数据包的概念以及可用于编解码器的另一 RTCP 消息，称为反馈消息（Feedback Message，FB）。a=rtcp-fb SDP 属性用于指示要使用的 FB 消息。例如 ack（肯定确认）和 nack（否定确认）。AVPF 配置文件可与 AVP 配置文件互操作。但是，由于缺少对最佳可用加密的支持，SAVP 无法与 AVP 配置文件互操作。

12.1.5 “通过一个端口多路传输 RTP 数据和控制数据包”[RFC5761]

此文档 [RFC5761] 说明如何在同一端口中多路传输 RTP 和 RTCP。此设计旨在简化 NAT 遍历，最大限度减少需要完成的“打洞”次数。此文档介绍了一种在 SDP 中使用属性

a=rtcp-mux 协商此事务的方法。

12.1.6 “用于混合器到客户端音频级别指示的实时传输协议标头扩展项” [RFC6465]

RFC 6465 [RFC6465] 可由 WebRTC 中的混合器使用，以向 WebRTC 用户代理指示混合音频会议中的音频级别。音频媒体混合器接收多个 RTP 流，并将其合并为单个流。混合器通常会实施一种混合策略，例如混合音量最大的三个活动扬声器。RTP 数据包可包含 CSRC（参与源标识符），用于标识最终参与混合数据包的各方。此 RTP 标头扩展项在上述信息的基础上增加了混合中包含的每个参与方的音频级别。音频级别编码为 7 位的 dBov，这是相对于系统过载点（最大响度）的分贝级别。关于是否提供此 RTP 标头扩展项，可通过 [RFC5285] 中说明的方法进行协商。例如，可根据与会者名册呈现此信息，以标识活动扬声器。

对于是否支持此 RTP 标头扩展项，由 SDP 的 a=extmap 属性中是否存在 urn:ietf:params:rtp-hdrext:csrc-audio-level 来指示。

12.1.7 “用于客户端到混合器音频级别指示的实时传输协议标头扩展项” [RFC6464]

此规范 [RFC6464] 可用于简化 WebRTC 会议中的混合器操作。通过此 RTP 扩展项，与会者可指示发送至混合器的数据包的音频级别。此信息非常有用，能够使混合器快速选择要包含在混合中的流，或使媒体选择器快速选取要选择的流，而无需对媒体数据包进行解码。音频级别编码为 7 位的 dBov，这是相对于系统过载点（最大响度）的分贝级别。关于是否提供此 RTP 标头扩展项，可通过 [RFC5285] 中说明的方法进行协商。对于此信息，可使用 [RFC6465] 中说明的方法将其复制到混合数据包中。

对于是否支持此 RTP 标头扩展项，由 SDP 的 a=extmap 属性中是否存在 urn:ietf:params:rtp-hdrext:ssrc-audio-level 来指示。

12.1.8 “RTP 流的快速同步” [RFC6051]

在 RTP 会话开始时，RTP 发送方和 RTP 接收方之间存在一段同步时间。对于简单的两方会话，这段时间极短。但对于多方会话，这段时间可能会较长。此文档 [RFC6051] 讨论了同步中的各种问题，并重新定义了 RTCP 定时和新 FB（反馈，Feed Back）消息来加快此过程。在 WebRTC 会话中，如果一个集中化媒体选择器要通过单个对等连接转发来自大量与会者的媒体数据包，上述机制将会非常有用。

12.1.9 “RTP 重新传输有效负载格式” [RFC4588]

如果媒体流的延迟要求并不严格，可使用此技术来请求重新传输丢失的 RTP 数据包。

此时需要使用 SAVPF 配置文件，以便快速发送 RTCP FB（反馈）数据包。由于 WebRTC 是实时通信技术，需要确保极低的延迟，因此这一方法是否适用还很难说。

12.1.10 “采用反馈 RTP/AVPF 的 RTP 音频 – 视频配置文件中的编解码器控制消息”[RFC5104]

此文档 [RFC5104] 说明了如何使用 AVPF 配置文件发送编解码器控制消息（Codec Control Messages，CCM）。这些编解码器控制消息可用于 H.271 视频反向通道、完全内部请求（Full Intra Request，FIR）、临时最大媒体流位速率（Temporary Maximum Media Stream Bit Rate，TMMBR）和临时空间权衡。TMMBR 可用于拥塞控制，如 11.4.4 节中所述。FIR 由视频接收方在发生视频切换时使用，以请求视频发送方发送 I 帧。此文档还定义了用于 a=rtcp-fb 属性的 ccm 参数。例如，a=rtcp-fb:98 ccm fir 可用于指示对有效负载 98 支持 FIR CCM。

12.1.11 “TCP 友好速率控制：协议规范”[RFC5348]

此文档 [RFC5348] 说明了一种拥塞控制机制，可确保实时 UDP 通信与 TCP 流合理共享带宽。此机制依赖于接收方向发送方发出的有关数据包丢失率和往返时间（Round Trip Time，RTT）的反馈。发送方随后使用 TCP 吞吐量公式计算 TCP 吞吐量，并据此调整其传输速率。采用拥塞控制的 WebRTC 方法可能会使用此规范。

12.1.12 “用于 RTP 标头扩展项的常规机制”[RFC5285]

RTP 规范 [RFC3550] 允许使用 RTP 标头扩展项，但并未规定如何指示或协商它们，亦未规定每个 RTP 数据包如何才能使用多个扩展项。此文档 [RFC5285] 对 RTP 标头扩展项进行了划分，从而允许存在多个扩展项，其中还说明了如何使用 SDP a=extmap 属性来指示它们。如果使用了 [RFC6464] 和 [RFC6465] 等 RTP 标头扩展项，则需要在 WebRTC 中使用此标头扩展机制。

12.1.13 “结合使用可变位速率音频与安全 RTP 的指南”[RFC6562]

此文档 [RFC6562] 讨论了可变位速率（Variable Bit Rate，VBR）编码和加密媒体的问题。可变位速率音频数据包流的速率和大小不一，这可能会泄露信息内容的相关信息，即使在经过加密时也不例外。此文档讨论了如何使用 RTP 填充来防范此问题。PCM（G.711）是一种恒定位速率（Constant Bit Rate，CBR）编解码器，Opus 则可以采用 VBR 或 CBR 模式。

12.1.14 “支持缩减型实时传输控制协议：契机与后果”[RFC5506]

此文档 [RFC5506] 讨论了可在哪些情况下发送缩减型 RTCP 数据包（即非复合数据包）。a=rtcp-rsize SDP 属性用于指示支持缩减型 RTCP 数据包。这样可使用完整型 RTCP 数据包

所占的带宽百分比交换更多 RTCP 数据包。WebRTC 使用 RTCP 反馈来实现多种目的。

12.1.15 “安全实时传输协议中的标头扩展项加密”[RFC6904]

常规 SRTP（详见 10.2.3 节）虽然对 RTP 标头扩展项进行身份验证，但并不对它们进行加密。此文档 [RFC6904] 定义了如何加密 RTP 标头扩展项。如果在 WebRTC 中使用了诸如 12.1.6 节和 12.1.7 节中介绍的 RTP 标头扩展项并需要提供隐私保护，则说明此扩展项可由 WebRTC 使用。RTP 标头是否已加密由 a=extmap SDP 扩展项指定。

12.1.16 “RTP 控制协议规范名称 (CNAME) 选择指南”[RFC7022]

此规范 [RFC7022] 定义了一种新的算法来生成随机 CNAME，这些规范名称通过 RTCP 发送，用于在 RTP 会话中标识 RTP 端点。这对于在 WebRTC 中保护媒体隐私十分重要，因为 CNAME 可用于在各会话和应用程序中标识发送方。

12.2 会话描述协议

12.2.1 “SDP：会话描述协议”[RFC4566]

此规范 [RFC4566] 定义了会话描述协议（Session Description Protocol，SDP）的版本 0。在 WebRTC 中，SDP 会话描述用于表示媒体提议或应答，并使用 JSEP（详见 11.3.8 节）进行传输和操作。SDP 提供了一种从连接 IP 地址和端口、媒体类型、编解码器和配置信息方面描述媒体会话的方式。但是，WebRTC 中需要许多 SDP 扩展项（未在此 RFC 中定义）。SIP 使用 SDP 来协商会话的机制称为“提议 / 应答”协议。SDP 和 SDP 扩展项的语法使用 ABNF 定义。

下面显示了一段最精简的 SDP 会话描述，其中包括一个 IPv6 地址：

```
v=0
o=alice 2890844526 2890844526 IN IP4 client.digitalcodexllc.com
s=-
c=IN IP6 FF1E:AD32::72EF:8D21:B866
t=0 0
m=audio 49178 RTP/AVP 98
a=rtpmap:98 OPUS/48000
```

12.2.2 浏览器中的 WebRTC　SDP 示例

本节将探讨由 Chrome 和 Firefox 浏览器生成的实际 SDP 提议和应答。浏览器的版本号显示在 SDP 旁边。如果使用每种浏览器的更高版本，SDP 可能会略有不同。请访问 WebRTC 书籍网站了解最新信息。

在每一节中，将先列出 SDP，再给出 SDP 的说明。这些提议和应答由本书中的演示应

用程序（http://demo.webrtcbook.com）捕获。最终通过信令通道进行的 SDP 提议 / 应答交换和候选项交换列在视频窗口下方。SDP 编码为 JSON 对象，其开头为 {"type":"offer","sdp":", 结尾为 ","type":"offer"} 或 ","type":"answer"}（具体取决于它是提议还是应答）。如果 SDP 提议或应答不包含候选项，则意味着浏览器在使用缓慢型 ICE（详见 11.5.1 节），候选项将通过信令通道单独发送。

12.2.2.1 Chrome Windows 与 Firefox Mac 之间的会话

此会话是从运行在 Windows 之上的 Chrome 32 到运行在 Macintosh 之上的 Firefox 的会话。Chrome 支持缓慢型 ICE，但 Firefox 并不支持此功能，因此 Chrome 提议不会显示任何候选项。

由 Chrome 32 Windows 生成的 SDP 提议

```
v=0
o=- 3844610931104753979 2 IN IP4 127.0.0.1
s=-
t=0 0
a=msid-semantic: WMS bnybSNzYSljqzLw7pXPQaHCOoete5eYJXPkc
m=audio 1 RTP/SAVPF 109 0 8 101
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:8NNHURsRlhAjOw/h
a=ice-pwd:6nZhTKpRwfJIXYbnLFz7qq3K
a=fingerprint:sha-256 DF:8C:B7:71:80:C7:66:09:48:D3:52:9E:96:19:8D:38:9A:9F:9C:DE
:E0:62:C0:85:22:98:AA:07:3A:76:BA:BD
a=setup:active
a=mid:audio
a=sendrecv
a=rtcp-mux
a=rtpmap:109 opus/48000/2
a=fmtp:109 minptime=10
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000
a=maxptime:60
a=ssrc:2842059344 cname:CtQf9S3X07RCWXSp
a=ssrc:2842059344 msid:bnybSNzYSljqzLw7pXPQaHCOoete5eYJXPkc 7b2095ed-0fb3-42ff-
ac92-cd306e1ce6f0
a=ssrc:2842059344 mslabel:bnybSNzYSljqzLw7pXPQaHCOoete5eYJXPkc
a=ssrc:2842059344 label:7b2095ed-0fb3-42ff-ac92-cd306e1ce6f0
m=video 1 RTP/SAVPF 120
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:Adu8/KM8PcTGF0gm
a=ice-pwd:+TgOyZBGaTptyf3n2QI0R1H9
a=fingerprint:sha-256 DF:8C:B7:71:80:C7:66:09:48:D3:52:9E:96:19:8D:38:9A:9F:9C:DE
:E0:62:C0:85:22:98:AA:07:3A:76:BA:BD
a=setup:active
```

```
    a=mid:video
    a=sendrecv
    a=rtcp-mux
    a=rtpmap:120 VP8/90000
    a=rtcp-fb:120 ccm fir
    a=rtcp-fb:120 nack
    a=ssrc:3018698479 cname:CtQf9S3X07RCWXSp
    a=ssrc:3018698479 msid:bnybSNzYSljqzLw7pXPQaHCOoete5eYJXPkc 55832df1-abb9-4165-
b1da-ff5ebabcbede
    a=ssrc:3018698479 mslabel:bnybSNzYSljqzLw7pXPQaHCOoete5eYJXPkc
    a=ssrc:3018698479 label:55832df1-abb9-4165-b1da-ff5ebabcbede
    m=application 1 DTLS/SCTP 5000
    c=IN IP4 0.0.0.0
    a=ice-ufrag:w+hx91AyIpbIxE0+
    a=ice-pwd:VdyR8sgUk4+k0YC+nsD6NBPh
    a=fingerprint:sha-256 DF:8C:B7:71:80:C7:66:09:48:D3:52:9E:96:19:8D:38:9A:9F:9C:DE
:E0:62:C0:85:22:98:AA:07:3A:76:BA:BD
    a=setup:active
    a=mid:data
    a=sctpmap:5000 webrtc-datachannel 1024
```

SDP的开头为4个必填字段：版本（v=）、来源（o=）、主题（s=）和时间（t=）。版本设置为0，这是当前使用的唯一SDP版本。来源包含一系列字段，其中包括用户、会话ID、版本ID、网络类型（IN表示Internet）、地址类型（IP4表示IP版本4）和地址；奇怪的是，此地址为环回地址（127.0.0.1）。

接下来是一个SDP属性（a=），用于定义此SDP中使用的媒体流ID语义；在此示例中，其语义为WMS（详见11.5.3节），它表示WebRTC媒体流。接下来，此SDP提议包含三个用于建立数据通道的媒体行（m=），依次为音频（m=audio）、视频（m=video）和应用程序（m=application）。再后面是一个媒体级别连接数据字段（c=），它设置为一个无效的IP地址（人们常说的“黑洞”地址0.0.0.0）。这是因为，此会话描述中的这一地址及所有其他c=地址都不会被实际使用，只有ICE候选项中的IP地址和端口才用于建立会话。

第一个音频媒体行列出了端口号（1，同样不会被使用，因为实际端口位于a=candidate字段中）、扩展的安全音频/视频配置文件（RTP/SAVPF）（详见12.1.4节），随后列出了4个可能的编解码器（有效负载类型分别为109、0、8和101）。这些编解码器按首选顺序列出，依次为：Opus、PCM mu-law、PCM A-law和telephone-event（用于发送DTMF，详见8.3.1.3节）。PCM编解码器使用静态有效负载（范围为0～95），而Opus和telephone-event则使用动态有效负载（范围为96～127）。对于所有这4个编解码器，Chrome都包括一个RTP映射属性（a=rtpmap），不过此属性对静态有效负载而言为可选属性。Opus采用48 kHz采样频率，属于立体声编解码器，如a=rtpmap属性中的/2所示。对于收到的Opus数据包，将在格式有效负载属性（a=fmtp）中将最短分包时间设置为10毫秒。PCM采用标准8 kHz采样频率。对于telephone-event，将最长分包时间设置为60毫秒（a=maxtime:60）。a=rtcp属性中包括一个RTCP端口和IP地址属性，但后者却设置为无效的IP地址和端口，因此尚

不清楚此属性的用途。

其中还包括 ICE 用户名片段（a=ice-ufrag）和 ICE 密码（a=ice-pwd）。请注意，对于每个 m= 行，将包括不同的用户名和密码。但是，在采用绑定时，将只使用第一组用户名和密码。接下来是指纹（a=fingerprint），这是将用于建立 DTLS 连接的自签名证书的 SHA-256 散列。此字段编码为一系列由冒号分隔的十六进制位。由于 DTLS 是一种面向客户端 / 服务器连接的协议，因此必须通过协商来确定由哪一方开通连接。在本示例中，Chrome 指示它将担任主动角色（a=setup:active）并负责开通连接。接下来是媒体 ID 属性，用于将此媒体行标记为“audio”（音频）（a=mid:audio）。send/receive 属性指示此音频会话为双向会话（a=sendrecv）。RTP 多路复用属性指示将通过用于 RTP 的同一端口对 RTCP 进行多路传输（a=rtcp-mux）。此音频行的最后一组属性指示，对于此 SSRC（a=ssrc），将使用特定的 RTCP CNAME。媒体流 ID、媒体流标签和标签由附加 a=ssrc 属性设置。请注意，Chrome 似乎遵循 draft-ietf-mmusic-msid-00，其中建议将此信息置于 a=ssrc 属性之中。该草案的最新版本（详见 11.5.3 节）建议改用 a=msid 属性。

对于视频媒体行（m=video），只提供了一个编解码器 VP8，它采用动态有效负载 120。这里使用相同的 SAVPF 配置文件。此外，还有一个媒体 ID“video”，以及用于指示双向媒体和 RTCP 多路复用的属性。对于此视频行，声明了三个 RTCP 反馈消息（a=rtcp-fb，详见 12.1.4 节）：

1. ccm：编解码器控制消息（Codec Control Messages，CCM），详见 12.1.10 节
2. fir：完全内部请求（Full Intra Request，FIR），详见 12.1.10 节
3. nack（否定确认），详见 12.1.4 节

与音频媒体行的处理方式一样，对于视频媒体行，也声明了 SSRC、CNAME、媒体流 ID、媒体流标签和标签。

最后一个媒体行用于数据通道（m=application），其中列出了虚拟端口（1）、DTLS/SCTP 传输和格式类型 (5000)。与前面一样，这里也选择了连接数据、ICE 用户名片段和密码、主动设置以及媒体 ID（值为“data”（数据））。最后一个属性为协议（webrtc-datachannel）的格式类型 5000 设置 SCTP 特性（a=sctpmap，详见 11.5.7 节），并设置最大通道数量（1024）。

下面是由 Firefox 生成的 SDP 应答。

由 Firefox Mac 生成的 SDP 应答

```
v=0
o=Mozilla-SIPUA-29.0a1 1653 0 IN IP4 0.0.0.0
s=SIP Call
t=0 0
a=ice-ufrag:c378ff48
a=ice-pwd:285af6088010426ea44b07a07a3458e1
a=fingerprint:sha-256 CA:6B:19:C3:50:48:FD:EE:98:9A:51:C1:DD:5D:35:E8:3C:15:CE:AE
:80:A0:08:C6:74:C7:B1:83:53:D6:59:43
```

```
    m=audio 57463 RTP/SAVPF 109 0 8 101
    c=IN IP4 109.144.215.158
    a=rtpmap:109 opus/48000/2
    a=ptime:20
    a=rtpmap:0 PCMU/8000
    a=rtpmap:8 PCMA/8000
    a=rtpmap:101 telephone-event/8000
    a=fmtp:101 0-15
    a=sendrecv
    a=setup:actpass
    a=candidate:0 1 UDP 2130379007 192.168.1.8 57463 typ host
    a=candidate:1 1 UDP 1694236671 109.144.215.158 57463 typ srflx raddr 192.168.1.8
rport 57463
    a=candidate:0 2 UDP 2130379006 192.168.1.8 49692 typ host
    a=candidate:1 2 UDP 1694236670 109.144.215.158 49692 typ srflx raddr 192.168.1.8
rport 49692
    a=rtcp-mux
    m=video 49912 RTP/SAVPF 120
    c=IN IP4 109.144.215.158
    a=rtpmap:120 VP8/90000
    a=sendrecv
    a=rtcp-fb:120 nack
    a=rtcp-fb:120 nack pli
    a=rtcp-fb:120 ccm fir
    a=setup:actpass
    a=candidate:0 1 UDP 2130379007 192.168.1.8 49912 typ host
    a=candidate:1 1 UDP 1694236671 109.144.215.158 49912 typ srflx raddr 192.168.1.8
rport 49912
    a=candidate:0 2 UDP 2130379006 192.168.1.8 56941 typ host
    a=candidate:1 2 UDP 1694236670 109.144.215.158 56941 typ srflx raddr 192.168.1.8
rport 56941
    a=rtcp-mux
    m=application 59328 DTLS/SCTP 5000
    c=IN IP4 109.144.215.158
    a=sctpmap:5000 webrtc-datachannel 16
    a=setup:actpass
    a=candidate:0 1 UDP 2130379007 192.168.1.8 59328 typ host
    a=candidate:1 1 UDP 1694236671 109.144.215.158 59328 typ srflx raddr 192.168.1.8
rport 59328
```

前4个字段（版本、来源、主题和时间）是SDP中的必填字段。来源字段使用“黑洞”IP地址0.0.0.0。主题设置为“SIP Call”（SIP呼叫），这可能是因为原始Mozilla RTC栈还包括SIP信令栈。接下来的三个属性是会话级属性，因为它们出现在第一个媒体行之前；它们分别设置ICE用户名片段和密码，以及用于DTLS的自签名公钥的SHA-256散列。对于此会话描述中的所有媒体行，这些值都相同。

此应答也有三个媒体行，因为它包含的m=行数量必须与提议相同。第一个媒体行是音频，其中包含一个非零端口号（57463），表示此媒体类型已被接受。这里使用了相同的SAVPF配置文件，并包括一组相同的有效负载类型。可用于媒体会话的有效负载类型由应

答中包括的集合给出，并且是所提议的集合的子集。在本示例中，会话期间可随时使用所有这 4 个有效负载。应答可包括提议中没有的附加有效负载类型，但由于它们没有包括在提议中，因此如不进行另一次提议 / 应答交换，就无法在此会话中使用。另请注意，应答中列出的动态有效负载类型与应答中的编号匹配（例如，109 表示 Opus，101 表示 telephone-event）。应答可使用不同的动态有效负载类型，前提是它们映射至相同的有效负载（例如，可使用 108 表示 Opus 并使用 103 表示 telephone-event）并且完全有效。此浏览器为接收数据而请求的分包时间为 20 毫秒（a=ptime:20）。连接数据地址（c=）字段包含服务器反射地址，但不会使用此地址；实际上，ICE 使用 a=candidate 候选项，详见 9.2 节。此音频会话为双向会话（a=sendrecv），DTLS 的方向性设置为主动或被动（a=setup:actpass）。由于提议为主动行为，这意味着此浏览器将在当前会话中设置为被动。接下来的 4 个属性为 ICE 候选项、RTP 的主机和服务器反射候选项，以及 RTCP 的主机和服务器反射候选项。首选设置是通过用于 RTP 的相同端口对 RTCP 进行多路传输，原因在于 a=rtcpmux。由于两个浏览器都支持绑定和 RTP/RTCP 多路复用，因此将只使用第一组候选项。

视频媒体行包含一个非零端口（49912），表示此媒体类型已被接受，并且编解码器将是 VP8（有效负载为 120）。同样，连接数据设置为服务器反射地址，并且视频为双向（发送 / 接收）会话。支持四种 RTCP 反馈消息：nack、pli（图片损失指示）、ccm 和 fir。同样，连接设置为主动或被动，并为 RTP 和 RTCP 提供了另外 4 个 ICE 候选项。由于两个浏览器都支持绑定，因此这些候选项也不会被使用。

最后一个媒体行接受具有非零端口（59328）的数据通道。SCTP 映射指示最多支持 16 个数据通道。此外，还包括另外两个未使用的 ICE 候选项。

请注意，虽然 Firefox 支持绑定，但 SDP 中并未指示这一点。

12.2.2.2 Chrome OS 与 Chrome Windows 之间的会话

此会话是从运行在 Chromebook 之上的 Chrome OS 32 到运行在 Windows 之上的 Chrome 32 的会话。两个浏览器都支持缓慢型 ICE，因此 SDP 中没有候选项。

由 Chrome OS 32 生成的 SDP 提议

由 Chrome OS 32 生成的此 SDP 提议与 Chrome Windows 生成的 SDP 提议类似，但也存在一些差别。会话级别属性 a=group:BUNDLE 指示此浏览器支持 BUNDLE 分组框架。BUNDLE 组包括所有这三个媒体行，在 a=mid 属性中分别标记为 audio、video 和 data。

请注意，这三个媒体行都使用同一（无效）端口 1 和无效的（黑洞）IP 地址 0.0.0.0。

a=extmap 属性中存在 urn:ietf:params:rtp-hdrext:ssrc-audio-level，这表示支持用于客户端到混合器音频级别指示的 RTP 扩展项（详见 12.1.7 节）。

a=ice-options:google-ice 属性指示，Chrome 支持更早版本的 ICE。如果 SDP 提议中没有此属性，则表示将使用标准（RFC 5245）ICE。

```
v=0
o=- 5476117230252359250 2 IN IP4 127.0.0.1
```

```
    s=-
    t=0 0
    a=group:BUNDLE audio video data
    a=msid-semantic: WMS yKbRQTdzN2DG8YAUsKNIJHOb0xC3ELyRIYoH
    m=audio 1 RTP/SAVPF 111 103 104 0 8 106 105 13 126
    c=IN IP4 0.0.0.0
    a=rtcp:1 IN IP4 0.0.0.0
    a=ice-ufrag:6faThXCg8GyVLdoE
    a=ice-pwd:zyOJiitDHxS/jBmwi8zfoXWo
    a=ice-options:google-ice
    a=fingerprint:sha-256
    DF:8C:B7:71:80:C7:66:09:48:D3:52:9E:96:19:8D:38:9A:9F:9C:DE:E0:62:C0:85:22:98:AA
:07:3A:76:BA:BD
    a=setup:actpass
    a=mid:audio
    a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
    a=sendrecv
    a=rtcp-mux
    a=crypto:1 AES_CM_128_HMAC_SHA1_80
    inline:xfx/v+memobaUvfbGkYAL9N6JEB6/Et2tpdNwc4P
    a=rtpmap:111 opus/48000/2
    a=fmtp:111 minptime=10
    a=rtpmap:103 ISAC/16000
    a=rtpmap:104 ISAC/32000
    a=rtpmap:0 PCMU/8000
    a=rtpmap:8 PCMA/8000
    a=rtpmap:106 CN/32000
    a=rtpmap:105 CN/16000
    a=rtpmap:13 CN/8000
    a=rtpmap:126 telephone-event/8000
    a=maxptime:60
    a=ssrc:2142953128 cname:NPMnExJy19ESBB9g
    a=ssrc:2142953128 msid:yKbRQTdzN2DG8YAUsKNIJHOb0xC3ELyRIYoH
    18e84811-abfb-47b2-b3d4-66c3295f5991
    a=ssrc:2142953128 mslabel:yKbRQTdzN2DG8YAUsKNIJHOb0xC3ELyRIYoH
    a=ssrc:2142953128 label:18e84811-abfb-47b2-b3d4-66c3295f5991
    m=video 1 RTP/SAVPF 100 116 117
    c=IN IP4 0.0.0.0
    a=rtcp:1 IN IP4 0.0.0.0
    a=ice-ufrag:6faThXCg8GyVLdoE
    a=ice-pwd:zyOJiitDHxS/jBmwi8zfoXWo
    a=ice-options:google-ice
    a=fingerprint:sha-256 DF:8C:B7:71:80:C7:66:09:48:D3:52:9E:96:19:8D:38:9A:9F:9C:DE
:E0:62:C0:85:22:98:AA:07:3A:76:BA:BD
    a=setup:actpass
    a=mid:video
    a=extmap:2 urn:ietf:params:rtp-hdrext:toffset
    a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
    a=sendrecv
    a=rtcp-mux
    a=crypto:1 AES_CM_128_HMAC_SHA1_80
    inline:xfx/v+memobaUvfbGkYAL9N6JEB6/Et2tpdNwc4P
```

```
    a=rtpmap:100 VP8/90000
    a=rtcp-fb:100 ccm fir
    a=rtcp-fb:100 nack
    a=rtcp-fb:100 goog-remb
    a=rtpmap:116 red/90000
    a=rtpmap:117 ulpfec/90000
    a=ssrc:3251765476 cname:NPMnExJy19ESBB9g
    a=ssrc:3251765476 msid:yKbRQTdzN2DG8YAUsKNIJHOb0xC3ELyRIYoH
    c81ec094-ff9f-4c0b-862f-408851892f1d
    a=ssrc:3251765476
    mslabel:yKbRQTdzN2DG8YAUsKNIJHOb0xC3ELyRIYoH
    a=ssrc:3251765476 label:c81ec094-ff9f-4c0b-862f-408851892f1d
    m=application 1 DTLS/SCTP 5000
    c=IN IP4 0.0.0.0
    a=ice-ufrag:6faThXCg8GyVLdoE
    a=ice-pwd:zyOJiitDHxS/jBmwi8zfoXWo
    a=ice-options:google-ice
    a=fingerprint:sha-256 DF:8C:B7:71:80:C7:66:09:48:D3:52:9E:96:19:8D:38:9A:9F:9C:DE
:E0:62:C0:85:22:98:AA:07:3A:76:BA:BD
    a=setup:actpass
    a=mid:data
    a=sctpmap:5000 webrtc-datachannel 1024
    ","type":"offer"}
```

此提议中没有ICE候选项，而是利用缓慢型ICE；借助缓慢型ICE，将生成一系列候选项并通过信令通道与另一端的浏览器进行候选项交换。

有效负载类型116表示red（针对音频的RFC 2198）或冗余视频；有效负载类型117表示ulpfec，即非均匀级保护前向错误纠正（Uneven Level Protection Forward Error Correction，ulpfec)(RFC 5109)。

由Chrome 32 Windows生成的SDP应答

以下是来自Windows Chrome浏览器的SDP应答。同样，此SDP不包含任何ICE候选项。ICE候选项将在SDP生成之后缓慢加入。

```
    v=0
    o=- 6377299728859741416 2 IN IP4 127.0.0.1
    s=-
    t=0 0
    a=group:BUNDLE audio video data
    a=msid-semantic: WMS B45cm6lwyRNVoE1zMlkijpbWUfL3rzNqscSq
    m=audio 1 RTP/SAVPF 111 103 104 0 8 106 105 13 126
    c=IN IP4 0.0.0.0
    a=rtcp:1 IN IP4 0.0.0.0
    a=ice-ufrag:wC9kk4acDp3y80J2
    a=ice-pwd:IaFjOBjaCJB9+JG4Amr33Zl5
    a=fingerprint:sha-256 D4:31:EA:41:3E:B5:73:CB:DE:49:B5:E7:52:7F:0E:66:DE:D7:1D:41
:2B:DF:D8:94:39:42:76:E7:CE:BF:D8:45
    a=setup:active
    a=mid:audio
    a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
```

```
    a=sendrecv
    a=rtcp-mux
    a=rtpmap:111 opus/48000/2
    a=fmtp:111 minptime=10
    a=rtpmap:103 ISAC/16000
    a=rtpmap:104 ISAC/32000
    a=rtpmap:0 PCMU/8000
    a=rtpmap:8 PCMA/8000
    a=rtpmap:106 CN/32000
    a=rtpmap:105 CN/16000
    a=rtpmap:13 CN/8000
    a=rtpmap:126 telephone-event/8000
    a=maxptime:60
    a=ssrc:3453958609 cname:145rMDOjvq7YHlyj
    a=ssrc:3453958609 msid:B45cm6lwyRNVoE1zMlkijpbWUfL3rzNqscSq
    905731da-ee01-4314-9577-05a4f1d31101
    a=ssrc:3453958609
    mslabel:B45cm6lwyRNVoE1zMlkijpbWUfL3rzNqscSq
    a=ssrc:3453958609 label:905731da-ee01-4314-9577-
    05a4f1d31101
    m=video 1 RTP/SAVPF 100 116 117
    c=IN IP4 0.0.0.0
    a=rtcp:1 IN IP4 0.0.0.0
    a=ice-ufrag:wC9kk4acDp3y80J2
    a=ice-pwd:IaFjOBjaCJB9+JG4Amr33Zl5
    a=fingerprint:sha-256 D4:31:EA:41:3E:B5:73:CB:DE:49:B5:E7:52:7F:0E:66:DE:D7:1D:41
:2B:DF:D8:94:39:42:76:E7:CE:BF:D8:45
    a=setup:active
    a=mid:video
    a=extmap:2 urn:ietf:params:rtp-hdrext:toffset
    a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
    a=sendrecv
    a=rtcp-mux
    a=rtpmap:100 VP8/90000
    a=rtcp-fb:100 ccm fir
    a=rtcp-fb:100 nack
    a=rtcp-fb:100 goog-remb
    a=rtpmap:116 red/90000
    a=rtpmap:117 ulpfec/90000
    a=ssrc:379840342 cname:145rMDOjvq7YHlyj
    a=ssrc:379840342 msid:B45cm6lwyRNVoE1zMlkijpbWUfL3rzNqscSq b69d0186-c4bc-4980-
bdae-b4729e38eebe
    a=ssrc:379840342 mslabel:B45cm6lwyRNVoE1zMlkijpbWUfL3rzNqscSq
    a=ssrc:379840342 label:b69d0186-c4bc-4980-bdae-b4729e38eebe
    m=application 1 DTLS/SCTP 5000
    c=IN IP4 0.0.0.0
    a=ice-ufrag:wC9kk4acDp3y80J2
    a=ice-pwd:IaFjOBjaCJB9+JG4Amr33Zl5
    a=fingerprint:sha-256 D4:31:EA:41:3E:B5:73:CB:DE:49:B5:E7:52:7F:0E:66:DE:D7:1D:41
:2B:DF:D8:94:39:42:76:E7:CE:BF:D8:45
    a=setup:active
    a=mid:data
    a=sctpmap:5000 webrtc-datachannel 1024
```

音频媒体行不仅包含 Opus、G.711 mu-law 和 A-law，还包含许多其他有效负载类型，例如采用速率为 16k 和 32k 的 ISAC，以及采样速率为 16k 和 32k 的柔和噪音（Comfort Noise，CN）。

视频媒体行包含一个 ICE 选项属性 google-ice。另外，此应答中还指示了两个 RTP 标头扩展项：toffset（传输时间偏移，在 RFC 5450 中定义）和试验性的 abs-send-time（绝对发送方时间，在 [draft-alvestrand-rmcat-remb] 中定义）。此视频媒体行还包含一个现已弃用的属性 a=crypto；如果使用 SDP 安全描述对 SRTP 加密（现在则强制使用 DTLSSRTP），则使用该属性。支持的 RTCP 反馈消息包括 ccm、fir、nack 和 goog-remb。此外，还支持 RED 和 ULPFEC。

12.2.3 “用于 RTP 控制协议带宽的会话描述协议带宽修饰符”[RFC3556]

此规范 [RFC3556] 定义了新的 SDP 带宽修饰符，适用于为 RTCP 指定带宽。通常情况下，RTCP 带宽的上限为总带宽的 5%。通过此规范中定义的 b=RS 和 b=RR 字段，可直接分别指定 RTCP 发送方带宽和 RTCP 接收方带宽。请注意，b=CT 和 b=AS 在 [RFC4566] 中定义，分别表示会议的总带宽和特定于应用程序的带宽。

12.2.4 “会话描述协议中特定于源的媒体属性”[RFC5576]

此规范 [RFC5576] 允许在 SDP 中指定参与流的各个媒体源的属性。请注意，术语“媒体流”的含义略微有些混乱。在某些上下文中，它是指 SDP 中由 m= 行定义的媒体对象。在其他上下文中，它是指 RTP 数据包的来源。在 WebRTC 中，同一个 m= 行可能会关联多个媒体源，这可以是来自同一用户的多个流，也可以是参与流的多个用户。此规范将媒体源定义为 RTP 中的 SSRC。此规范定义了 a=ssrc 属性（Attribute），通过该属性可声明 SSRC 的特性（Property）。其中还定义了 CNAME（cname）、前 SSRC（previous-ssrc）和格式特定的参数等特性。

12.2.5 “在 SDP 中协商通用图像属性”[RFC6236]

此规范 [RFC6236] 定义了用于协商图像属性的 a=imageattr SDP 属性。请考虑以下示例：

```
a=imageattr:97 send [x=800,y=640,sar=1.1,q=0.6]
[x=480,y=320] recv [x=330,y=250]
```

此属性为有效负载 97 设置了发送和接收图像的大小（以像素为单位）。就发送而言，这里提供了两个可能的图像大小。第一个图像大小的存储宽高比（Storage Aspect Ration，sar）为 1.1，首选项值为 0.6。第二图像大小具有默认 sar（1.0 表示方形像素），且默认首选项值为 0.5。此外，还可以设置可接受的图片宽高比（Picture Aspect Ratio，par）值的范围。

12.3 NAT 遍历 RFC

12.3.1 “交互式连接建立：用于提议/应答协议的网络地址转换器遍历协议”[RFC5245]

如第 9 章所述，WebRTC 使用交互式连接建立（Interactive Connectivity Establishment，ICE）[RFC5245] 来进行网络地址转换器（NAT）遍历和媒体授权。ICE 是一种标准化的“打洞”协议，此技术由游戏领域开发，用于在两个位于 NAT 之后的主机之间建立对等连接。每个主机都会收集潜在的地址候选项：本地地址（从网卡读取）、反射地址（从 STUN 服务器确定）和中继地址（从 TURN 服务器或其他媒体中继获取），如图 9.1 所示。这些候选项先经过优先排序，再编码为 SDP 中的 a=candidate 行，然后通过位于公共 Internet 中的服务器（称为“集合服务器”）进行交换。接下来，两个主机大体同时开始发送测试数据包（有时称为“打洞数据包”）。当它们尝试向对方的候选项地址发送测试数据包时，这些数据包就会创建 NAT 映射和过滤器规则。在许多情况下，通过发送几次测试数据包，即可获取穿过 NAT 的端到端路径，此连接将在会话的整个持续期间内使用。在某些情况下，由于 NAT 或防火墙十分严格，无法建立对等连接。此时，将改用 TURN 媒体中继地址。采用 ICE 或其他类似打洞方式的服务提供商在未发表的统计资料中指出，可获取直接连接的概率高达 85%。目前，在 Web 服务器中，可按照类似于配置 Web 代理的方式（并出于类似的原因）配置 TURN 服务器地址，以便启用防火墙遍历，详见 9.2 节。

只有当另一端的主机收到经过身份验证的答复打洞数据包时，才会使用 ICE 中的候选项地址，因此这种设计可以提供所需的媒体授权。只有当候选地址需要并主动尝试建立会话时，该候选项才会成功并由会话使用。这可以防止“语音锤”攻击；在这种攻击中，攻击者会提供另一个主机的候选地址，企图向该主机滥发大量通信。

连接地址候选项和地址类型由 SDP 属性 a=candidate 承载。例如：

```
a=candidate:2 1 UDP 1694498815 192.0.2.3 45664 typ srflx raddr 10.0.1.1 rport 8998
```

此规范定义了值 host、srflx、prflx 和 relay 来分别表示主机候选项、服务器反射候选项、对等端反射候选项和中继候选项。

12.3.2 “对称 RTP/RTP 控制协议 (RTCP)”[RFC4961]

此文档 [RFC4916] 定义了对称 RTP 和 RTCP，并针对其适用场合提供了指南。在双向 RTP 会话中，如果通过同一 UDP 端口发送和接收数据包，则说明 RTP 是对称的。这对于遍历 NAT 及通过 TURN 和其他媒体中继（例如 SBC 提供的媒体中继）进行遍历具有重要意义。WebRTC 中的所有媒体都是对称的。

12.4 编解码器

12.4.1 “Opus 音频编解码器的定义”[RFC6716]

Opus [RFC6716] 是适用于音频和音乐的 Internet 低延迟编解码器。Opus 借鉴了 Skype SILK [SILK] 编解码器和开源受限能量重叠转换（Constrained Energy Lapped Transform，CELT）[CELT] 编解码器中的元素和技术。Opus 是一种极为灵活的解决方案，具体体现为：支持 6-510 kb/s 的位速率；支持恒定或可变位速率；支持 8-48 kHz 的取样率；支持语音和音乐；支持单声道和立体声；支持 2.5 ～ 60 毫秒的帧大小；并支持浮点或定点算法。Opus 还具有十分精良的数据包丢失隐藏（Packet Loss Concealment，PLC）功能，即使在数据包丢失期间，也能提供良好的质量。Opus 的 RTP 有效负载在 [draft-ietf-payloadrtp- opus] 中定义。

12.4.2 “VP8 数据格式和解码指南”[RFC6386]

VP8 [RFC6386] 是 WebRTC 中使用的一种开源视频编解码器。VP8 的 RTP 有效负载在 [draft-ietf-payload-vp8] 中定义。

12.5 信令

作为会话初始协议传输机制的 WebSocket 协议”[RFC7118]

此文档 [RFC7118] 定义了会话初始协议（SIP）的 WebSocket 传输机制。通过 WebSocket（详见 10.2.2 节），Web 浏览器可打开一个通向 Web 服务器的新连接。虽然此规范并非 WebRTC 所必需的规范，但它允许将会话初始协议（Session Initiation Protocol，SIP）用作信令协议，因而具有很大的关联性。例如，SIP 用户代理（User Agent，UA）栈以 JavaScript 编写并由 Web 服务器下载。随后，SIP UA 将打开一个通向 SIP 代理服务器的新 WebSocket 连接。来自 SIP 信令通道的媒体将使用常规 WebRTC 方法（例如对等连接）来建立媒体会话。此规范定义了新的 Via 传输令牌 WS（WebSocket）以及新的 SIP URI 传输参数 ws（WebSocket）和 wss（Secure WebSocket）；后者使用 TLS 传输机制。

请注意，Web 服务器之间的 SIP 信令（如图 1.4 中的 WebRTC 梯形所示）很可能不使用 WebSocket 传输机制，而是使用 TCP 或 UDP 等常规 SIP 传输机制。

12.6 参考资料

[RFC3550] http://tools.ietf.org/html/rfc3550
[RFC4568] http://tools.ietf.org/html/rfc4568

[RFC3551] http://tools.ietf.org/html/rfc3551
[RFC3711] http://tools.ietf.org/html/rfc3711
[RFC6188] http://tools.ietf.org/html/rfc6188
[RFC5764] http://tools.ietf.org/html/rfc5764
[RFC5124] http://tools.ietf.org/html/rfc5124
[RFC4585] http://tools.ietf.org/html/rfc4585
[RFC5761] http://tools.ietf.org/html/rfc5761
[RFC6465] http://tools.ietf.org/html/rfc6465
[RFC5285] http://tools.ietf.org/html/rfc5285
[RFC6464] http://tools.ietf.org/html/rfc6464
[RFC6051] http://tools.ietf.org/html/rfc6051
[RFC4588] http://tools.ietf.org/html/rfc4588
[RFC5104] http://tools.ietf.org/html/rfc5104
[RFC5348] http://tools.ietf.org/html/rfc5348
[RFC5285] http://tools.ietf.org/html/rfc5285
[RFC6562] http://tools.ietf.org/html/rfc6562
[RFC5506] http://tools.ietf.org/html/rfc5506
[RFC6904] http://tools.ietf.org/html/rfc6904
[RFC7022] http://tools.ietf.org/html/rfc7022
[RFC4566] http://tools.ietf.org/html/rfc4566
[draft-alvestrand-rmcat-remb] http://tools.ietf.org/html/draft-alvestrand-rmcat-remb
[RFC3556] http://tools.ietf.org/html/rfc3556
[RFC5576] http://tools.ietf.org/html/rfc5576
[RFC6236] http://tools.ietf.org/html/rfc6236
[RFC5245] http://tools.ietf.org/html/rfc5245
[RFC4961] http://tools.ietf.org/html/rfc4961
[RFC6716] http://tools.ietf.org/html/rfc6716
[RFC6386] http://tools.ietf.org/html/rfc6386
[SILK] http://developer.skype.com/silk
[CELT] http://www.celt-codec.org
[draft-ietf-payload-vp8] http://tools.ietf.org/html/draft-ietf-payload-vp8
[RFC7118] http://tools.ietf.org/html/rfc7118

Chapter 13 第 13 章

安全和隐私

本章将介绍和探讨 WebRTC 安全的诸多方面。其中包括：

1）恶意网站可能针对浏览器发起的新攻击和破坏行为。

2）WebRTC 建立的会话是否安全，以及可能遭到哪些类型的攻击。

3）WebRTC 是否会在用户浏览时侵犯用户隐私。

4）允许 WebRTC 会话跨越企业边界是否安全。

有关 WebRTC 方面的安全威胁和概念的完整讨论，请参见 [draft-ietf-rtcweb-security] 和 [draft-ietf-rtcweb-security-arch]。

学习本章的前提是，基本熟悉隐私和身份验证等安全概念，以及加密、数字签名和证书等安全方法。有关这些主题的深入讲解，请参考 [APPLIED-CRYPTO]。

13.1 浏览器安全模型

对 WebRTC 安全的任何讨论都必须从基本的 Web 浏览器安全模型开始。在此模型中，用户必须信任其 Web 浏览器。恶意浏览器可能会将用户重定向到他们不想访问的网页，记录浏览历史和数据并泄露给未经授权的第三方，或者在未告知用户的情况下访问计算机的麦克风和网络摄像头。简而言之，如果协议或 API 提供的安全措施被忽略或错误实施，就不再具有任何价值。请注意，用户安装的任何软件都存在这种隐患。

然而，网站并不一定值得信任。用户必须能够浏览恶意网站，而不受到任何伤害。虽然用户不会访问他们认为可能有恶意的网站，但结果总是防不胜防。例如，用户可能点击一个缩短的 URL 链接，结果进入恶意网站。又或者，一个链接或网站将用户连接或重定向

到恶意网站。当用户访问网站时，该站点可能会控制屏幕、显示任意图像或文本，甚至可能发起视频或音频流。当然，用户可以直接关闭浏览器选项卡或窗口，结束这一切。如果网站尝试下载文件，浏览器将禁止此行为，除非用户单击相关对话框予以授权。某些操作系统还会在用户执行使用浏览器下载的软件之前，向用户发出警告。所有这些功能都旨在帮助保护用户免受恶意网站侵害。

13.1.1 WebRTC 权限

WebRTC 提供了一些类似的保护。特别是在提供对用户麦克风或摄像头的访问权限之前，WebRTC 要求浏览器确认已得到用户的同意。对于许多浏览器而言，这意味着在提示用户并获得授权之前，意图访问用户麦克风或摄像头的 WebRTC 应用程序不会获得访问权限。此授权对话框由浏览器自行处理，而不受 JavaScript 控制。图 13.1 和图 11.1 中显示了一个浏览器确认示例。每次会话都要求提供这种权限——除非用户想要参与实时会话，否则应用程序无法随时访问麦克风或摄像头。另外，这种权限是按域授予的。这一点很重要，因为给定的 Web 窗口视图中经常会呈现来自许多不同网站的内容，包括其他画面、广告和跟踪软件。如果给定页面上的多个域希望访问此媒体，则每个域需要分别获得权限。这可以防止一个域获得授权被其他域盗用。

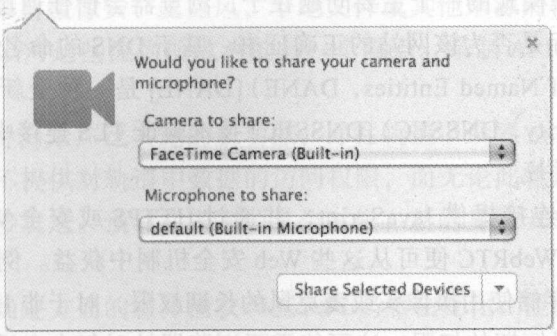


图 13.1 浏览器提示用户提供权限

有趣的是，仅建立一条数据通道的 WebRTC JavaScript 调用不需要用户许可。其背后的逻辑是，用户在访问网站时，已经许可该网站向浏览器发送任意数据，而这正是数据通道所允许的。只有获得用户媒体时才需要权限检查，而获得权限后的传输过程并不需要。

仅当浏览器可以安全地确定申请权限的网站时，强制授权才会有效。这是下一节要讨论的内容。

13.1.2 网站身份

在万维网的早期发展阶段，人们发现需要确定网站的身份并对网站会话进行加密和身份验证。安全套接字层（Secure Sockets Layer，SSL）应运而生，现已发展为 TLS 协议（详

见 10.2.8 节）。安全 Web 浏览在当今得到越来越普遍的运用，即浏览器和网站之间使用安全 HTTP（HTTPS）。过去，此技术主要在登录期间和执行敏感的财务事务时使用。然而，现在它成为默认的浏览方式，电子前沿基金会的 HTTPS Everywhere [HTTPS-EVERY] 等浏览器插件可轻松确保在 Web 浏览期间达到最高隐私级别。

如果使用安全浏览（利用 TLS 传输的安全 HTTP），而非普通浏览（只使用 TCP 传输的普通 HTTP），则可确定提供 WebRTC 服务或应用程序的网站的身份。通过开通连接时执行的 TLS 握手，浏览器便可以识别网站的身份。在这种方法中，网站会提供由证书颁发机构（Certificate Authority，CA）颁发的数字证书，其中 CA 是浏览器中的一个可信锚点（Trust Anchor）。浏览器预先配置了一组可信锚点证书颁发机构。这些证书颁发机构会基于特定域名的所有权和控制证明，向该域名的持有者颁发数字证书。例如，example.com 域的所有者可以从证书颁发机构购买数字证书（X.509 证书）。然后，example.com 域中的 Web 服务器便能够在 TLS 握手过程中使用该证书来证明自己位于 example.com 域中。浏览器会向证书颁发机构验证该证书，以确保其真实性。

如果身份得到证实，浏览器会显示挂锁或其他用户界面提示信息，以表明这是经过身份验证的安全 Web 会话。否则，浏览器会显示警告，并建议用户断开连接。登录 WiFi 重定向期间，经常会遇到这种类型的警告。

这种网站身份验证模式的一个主要问题在于，浏览器会信任通过验证的网站的任何证书。它无法得知这实际是否为该网站的正确证书。基于 DNS 的命名实体身份验证（DNS-Based Authentication of Named Entities，DANE）[DANE] 是一种全新的安全机制，它使用 DNS 安全（DNS Security，DNSSEC）[DNSSEC] 技术验证 TLS 连接中收到的数字证书是否为该网站使用的正确证书。

如果通过 HTTPS 连接提供 JavaScript，并通过 HTTPS 或安全 WebSocket（一种 WSS URI）提供信令通道，WebRTC 便可从这些 Web 安全机制中获益。例如，如果 WebRTC 网站使用 HTTPS，则可存储使用摄像头或麦克风的长期权限。对于非 HTTPS WebRTC 网站，只能在每次使用时获得短期的权限，这可能需要浏览器针对每个会话向用户发出提示，以确认用户是否同意。

13.1.3 浏览器用户身份

遗憾的是，上一节中所介绍的网站身份验证方法不能用于验证浏览器用户的身份。虽然浏览器可以在 TLS 握手期间出具数字证书，但浏览器通常没有安装数字证书。这是为了降低成本和方便使用，而且用户有多个身份，每个身份都需要单独的证书。例如，对于由可信证书颁发机构提供的低价证书，其每年的费用在 60 ～ 100 美元。这对于大多数服务和应用程序提供商而言，都是一项重大开支。

因此，Web 上一般使用其他方法，例如提示用户输入用户名 / 密码组合，或者使用以前经过身份验证的会话中的“Cookie”。有时还使用第三方身份识别服务。WebRTC 可重复使

用这些身份验证方法，本章稍后将讨论此内容。

13.2 新型 WebRTC 浏览器攻击

WebRTC 采用全新的 API、协议和信令通道，无疑会引入许多针对浏览器的新型潜在攻击。下面将介绍针对上述每一种要素的攻击。

13.2.1 API 攻击

WebRTC 引入了大量全新的 JavaScript API。其中每一个都可能成为攻击浏览器的全新途径。

我们已经谈到，JavaScript 代码需要得到用户同意才能申请访问用户的摄像头和麦克风。实际上，媒体捕获 API 不会提供针对 MediaStreamTrack 中的流动数据的访问权限，而只是提供可传递给其他页面元素的轨道句柄。然而，访问轨道上的流动数据相当简单。通过创建一个包含用户视频的 <canvas>，就可能有机会按固定间隔对该画布采样。如此可对用户的视频流应用一些有趣的 JavaScript 编码效果，但这意味着，如果用户同意访问摄像头，则暗示也可以访问来自摄像头的数据。

这样还可以获取来自摄像头、麦克风、本地文件或其他来源的任何流，并通过对等连接将其流式传输到可以对这些流为所欲为的另一个端点。录制 API 提供更为直接的访问，因而可以轻松保存流。

对此，目前有一种尚未广泛实施的新功能，即在轨道上设置 noaccess 标记。设置此标记后，浏览器可确保不提供对轨道中数据的访问权限，而无论此轨道用于画布中的视频元素，还是通过对等连接发送。预计在未来的某个时候，新的身份 API 将允许对轨道实施限制身份的访问。

目前，关于功能确定方面的媒体捕获和流 API 仍然在变化，但应用程序或许能够在用户同意之前获得设备名称，并获得关于用户同意访问的任何设备的详情。

用户必须谨慎地授予权限。经过一段时间后，浏览器的某些设置可能会让用户为特定网站提供较长期的权限，以访问用户的摄像头和麦克风。请记住，只要权限的持续时间超过用户操作设备的时间，就可能导致应用程序在用户不方便的时间使用设备，例如晚上或在设备旁边进行私下聊天时。

除了上述普遍性问题，以及因在 Web 应用程序中使用任何 JavaScript 所造成的威胁，API 应该不会带来全新的重大威胁。然而，这些 API 是全新的，因此始终有可能出现实施错误，导致这些 API 变身为攻击浏览器和 WebRTC 会话的潜在工具。

13.2.2 协议攻击

图 6.1 显示了 WebRTC 中存在的新协议。其中每个协议都为攻击者提供了新的机会。

例如，由于 WebRTC 使用 SDP，因此攻击者可能使用不正常的 SDP 对象，以试图摧毁浏览器。WebRTC 使用 RTP 进行媒体传输，因此可能有人基于 RTP 进行攻击。此外，在建立的会话期间，浏览器将利用编解码器对收到的音频或视频样本进行解码。攻击者可能会使用不正常的媒体样本，以试图破坏浏览器。

WebRTC 的全新体系结构还会强化其中部分攻击的破坏力。在 WebRTC 之前，浏览器仅与 Web 服务器通信。有了 WebRTC，浏览器可以直接与其他浏览器或其他设备建立对等连接。这意味着，浏览器不仅需要抵御恶意网站，还要防范恶意浏览器。一个非常安全的 WebRTC 应用程序可能会允许另一个浏览器通过对等连接执行攻击。

另一方面，恶意网站可能会利用 WebRTC 对其他主机发动拒绝服务攻击。图 13.2 中显示了一种潜在的攻击。

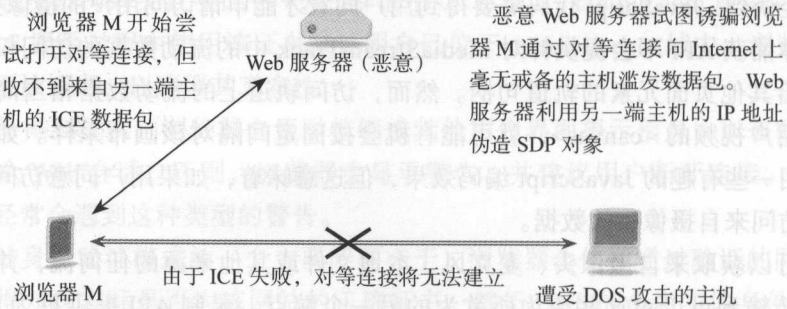


图 13.2 恶意 Web 服务器利用 WebRTC 发起 DOS 攻击

恶意 Web 服务器可能会试图诱骗一个浏览器向另一个主机发送高带宽流（例如高清视频）。幸运的是，WebRTC 在开始任何媒体流之前，都要求 ICE 指示已得到远程站点的同意。这样，Web 服务器就无法假冒这些 ICE 数据包，从而可阻止这类 DOS 攻击。不过，WebRTC 仍然可能会产生大量的 Internet 流量，给浏览器造成一种 DOS 攻击。例如，用户可能会授权 WebRTC 会话，而没有意识到将消耗巨大的带宽。例如，一个网站可能导致浏览器向服务器发起十个高清（High Definition，HD）视频流，这会直接降低其他流量，而用户还在奇怪为什么 Internet 连接如此缓慢。这可能会干扰该主机或共享通用基础架构的其他主机上的其他 Internet 活动。

ICE 是浏览器中实施的另一个协议，因此也可能利用 ICE 发起攻击，例如不正常的 ICE 数据包。另外，鉴于 ICE 打洞的性质，在建立对等连接的这一阶段，浏览器会接受和处理从任何目标传入的数据包。幸运的是，ICE 握手完成后，将基于特定的 IP 地址锁定会话。然而，ICE 中使用的恶意 STUN 或 TURN 服务器仍然可能会试图中断会话。

13.2.3 信令通道攻击

WebRTC 依赖信令通道来建立媒体和数据通道。如果信令通道或信令服务器受损，攻击者就能实施多种攻击行为。例如，攻击者可以阻止建立媒体或数据通道会话，也可以降

低连接的安全等级。攻击者还可以将连接重定向到另一个用户，或者通过一个中间人设备记录、插入或修改媒体或数据，或以其他方式干扰会话。因此，保护 WebRTC 信令通道非常重要。

WebRTC 信令通道的实施方式会影响可能招致的攻击。例如，如果使用 WebSocket，就可能发生针对 WebSocket 的潜在攻击。如果使用 SIP 或 Jingle，则可能出现这些协议实施方案中已知的和全新的漏洞。有关 SIP 的详细安全分析，请参考 [VOIP-SEC]。

Web 服务器使用同源安全模型来防止处理意外的 Web 通信。然而，从带宽和性能角度看，强制信令通过 Web 服务器并不是理想的方案。因此，对于使用跨来源资源共享（Cross-Origin Resource Sharing，CORS）[CORS] 的 WebRTC 信令，可放宽同源策略的要求。WebSocket（详见 10.2.2 节）允许放宽同源策略的要求。因此，SIP 代理、XMPP 服务器或 WebSocket 代理可以独立于 Web 服务器运行。

对于 XHR [XHR-API] 生成的 HTTP 请求，也可以使用 CORS，以便将 HTTP 信令消息路由至 Web 服务器以外的其他服务器。如果使用 HTTPS，则浏览器在发送信令信息之前，可先对此服务器进行身份验证。

13.3 通信安全

通信安全表示使用 WebRTC 建立的实时通信会话的安全。隐私保护和身份验证是 WebRTC 支持的两个重要安全属性。

13.3.1 通信隐私

隐私保护有助于防止第三方窃听语音或视频通信会话。通过端到端加密可提供隐私保护。在使用安全 RTP [RFC3711] 的 WebRTC 中，默认对音频和视频会话提供隐私保护。SRTP 使用对称密钥对媒体数据包加密和解密，并通常使用 AES 计数器模式加密密码。对于数据通道，将由 DTLS 提供加密。数据通道的数据还可以通过 JavaScript 加密和解密，因而又增加了一重安全保障。

图 13.3 显示了 SRTP 标头中经过加密和身份验证的部分。请注意，对于 RTP 标头，只进行身份验证，但不加密。这意味着，SRTP 数据包可被识别为 RTP，但如果没有密钥，就无法播放或录制媒体。通过允许检测 SRTP 流量，可以对网络应用策略，并提供服务质量（QoS）和监控服务。

SRTP 的安全严重依赖于密钥方法。SRTP 密钥方法有两种：一是通过信令通道发送密钥；二是在媒体路径中生成密钥。SDP 安全描述是前一种方法的示例，DTLS-SRTP 则是后一种方法的示例。

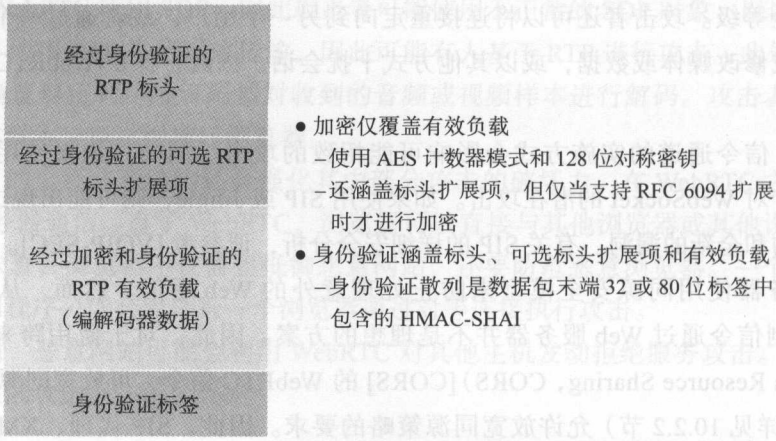


图 13.3　SRTP 数据包中经过加密和身份验证的部分

13.3.2　通过信令通道传输密钥

通过信令通道发送密钥时，信令通道必须满足一些安全要求。如果信令通道没有加密，则整个 SRTP 媒体会话将失去隐私保护。安全 HTTP（HTTPS）或安全 WebSocket（WSS）传输可为信令通道提供这种加密。此外，由于信令通常经过信令服务器，因此该信令服务器必须值得信任，并对其提供保护，防止其遭到破坏。特别是，当用于传输 SRTP 密钥的信令服务器进行日志记录时，必须对这些日志进行保护并加密，否则 SRTP 密钥就可能遭到泄露。

SDP 安全描述 [RFC4568] 的通俗叫法是 SDES，它是使用 SRTP 的 VoIP 和视频系统最常用的信令协议。SDP 安全描述在 a=crypto SDP 属性中包含 SRTP 主密钥和 salt。例如：

```
a=crypto:1 AES_CM_128_HMAC_SHA1_32 inline:a0RgdmcmVCspeWpVLFJhfHAQ
```

此代码将 SRTP 配置为使用 AES 计数器模式、128 位密钥和 32 位 HMAC SHA-1 身份认证标签。Inline 包含以 base 64 编码的密钥和 salt。请注意，WebRTC 中不允许使用 SDP 安全描述。

13.3.3　媒体路径中的密钥协议

媒体路径中的密钥协议不依赖于信令通道的安全性或信令服务器的可信性。实际上，这些密钥由媒体路径（通过用于最终媒体会话的同一传输 IP 地址和端口号）中的公钥运算生成。Diffie-Hellman（DH）密钥协议就是此类公钥运算的一个示例。DTLS-SRTP 是在 DTLS 握手期间执行此类运算的媒体路径密钥协议。DTLS 握手发生在 ICE 连接检查完成并为媒体会话选择候选项对之后。图 13.4 显示了 DTLS-SRTP 的运行过程。

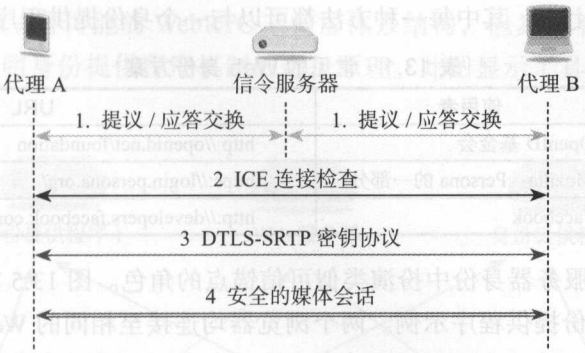


图 13.4 DTLS-SRTP 密钥协议的运行过程

13.3.4 身份验证

利用通信会话身份验证功能，可确定媒体数据包来自会话的另一方。身份验证标签内部包含的 HMAC 是 SRTP 媒体身份验证的一部分，如图 13.3 所示。

13.3.5 身份

目前，有若干方法可用于确定 WebRTC 会话中参与者的身份。两种方法都依赖于 DTLS-SRTP 密钥方法。如果浏览器拥有属于 PKI 的客户端证书，则可在 DTLS 握手期间传递并验证此证书。但拥有 PKI 浏览器证书的情况并不常见。

如果不使用 PKI 证书，则浏览器会使用常说的自签名证书。此数字证书并非证书颁发机构所颁发，因而不是信任链中的一部分。它本质上就是一个公钥容器，此公钥与 SRTP 密钥协商期间用于公钥运算的私钥相关联。该证书的指纹（散列函数的输出）包含在 SDP 提议或应答消息中。如果信令通道支持端到端完整性保护，则可以提供身份验证，但这不是很常见。

13.4 WebRTC 中的身份

WebRTC 中应用了两种身份。第一种是提供 WebRTC 应用程序的网站的身份。这种身份适用于任何类型的网站，而 WebRTC 只需使用基于浏览器的现有方法（详见 13.1.2 节）即可。另一种身份是位于对等连接另一端的最终用户的身份。此身份应 WebRTC 而生，对于浏览器来说是一种新生事物，因为大多数网站仅直接与 Web 服务器交互，而不是像 WebRTC 一样允许与其他浏览器直接交互。

WebRTC 定义了一种新的身份方案，这是在现有跨网站 ID 和单点登录方法基础之上的扩展。表 13.1 中列出了部分常见的基于 Web 的方法。[draft-ietf-rtcweb-security-arch] 中定义的身份提供程序（Identity Provider，IdP）是一种用于在 WebRTC 提供身份的广义 Web 身

份方案框架。从理论上讲，其中每一种方法都可以与一个身份提供程序一起使用。

表 13.1 常见的 Web 身份方案

方法	使用者	URL
OpenID	OpenID 基金会	http://openid.net/foundation
BrowserID	Mozilla、Persona 的一部分	https://login.persona.org/
Facebook 登录名	Facebook	http://developers.facebook.com/docs/facebook-login

身份提供程序在服务器身份中扮演类似可信锚点的角色。图 13.5 在 WebRTC 三角形中显示了一个简单的身份提供程序示例。两个浏览器均连接至相同的 Web 服务器，并且使用相同的身份提供程序。请注意，网站和身份服务之间不需要存在任何交互或配合，它们可以完全独立。这样一来，WebRTC 提供程序可以完全置身事外，不提供任何身份。

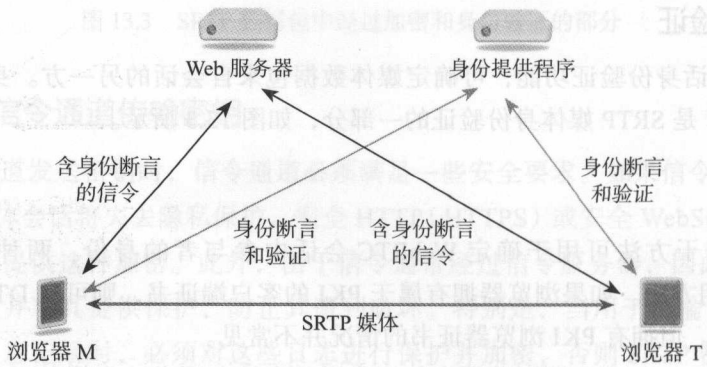


图 13.5 WebRTC 三角形中的身份提供程序

图 13.6 在 WebRTC 梯形中显示了更一般的身份提供程序示例。在此示例中，有两个独立的 Web 服务器，二者之间具有信令连接，允许在每个网页中建立浏览器间的对等连接。此外，还有两个独立的身份提供程序，每个都未与 Web 域关联。

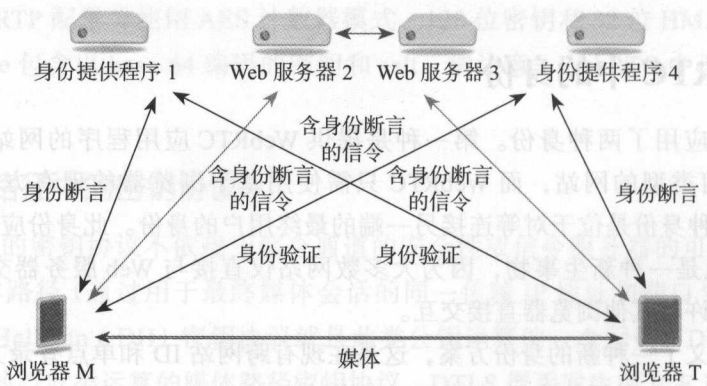
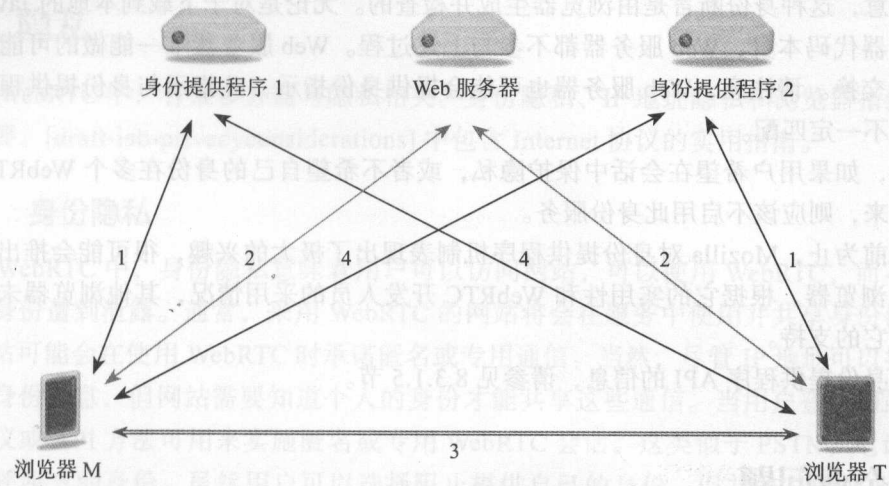


图 13.6 WebRTC 梯形中的 WebRTC 身份提供程序

图 13.7 显示了另一种可能的 WebRTC 三角形体系结构，但具有不同的身份提供程序。此体系结构将用来说明身份提供程序机制的运行原理。此图显示了具有两个身份提供程序的单一 Web 域。



1）用户登录自己的 IdP。浏览器接收材料以签署要使用的 SRTP 密钥材料。

2）通过信令通道发送密钥材料的签名。

3）使用 SRTP 密钥材料来协商 SRTP 媒体会话。

4）对照通过信令接收的签名，检查所收到的 SRTP 密钥材料，并由其他人的 IdP 进行验证。

图 13.7 在 WebRTC 中使用身份提供程序的步骤

在此示例中，浏览器 M 上的用户在身份提供程序 1 中具有账户，并已将他的浏览器配置为使用身份提供程序 1 作为自己的身份提供程序。该用户登录到身份提供程序 1。当浏览器 M 上的用户启动 WebRTC 会话时，浏览器将联系身份提供程序 1 以获取用于生成身份签名的短期证书。当收到证书后，浏览器 M 将通过用于建立对等连接的 SRTP 密钥材料生成一个数字签名。此签名通过信令通道随 SDP 对象发送至 Web 服务器。Web 服务器将 SDP 对象和身份签名转发至浏览器 T。

浏览器 T 上的用户在身份提供程序 2 中具有账户，并已将他的浏览器配置为使用身份提供程序 2 作为自己的身份提供程序。该用户登录到身份提供程序 2，收到 SDP 对象后，浏览器 T 便联系身份提供程序 2 以获取用于生成身份签名的短期证书。系统生成 SDP 对象，并通过将用于会话的 DTLS 指纹生成签名。二者通过信令通道进行发送，信令通道会将它们转发至浏览器 M。浏览器 M 和浏览器 T 开始 ICE 打洞，然后进行 DTLS 握手。建立

DTLS 后，每个浏览器将分别使用基于 SDP 对象的签名，检查用于建立 DTLS 连接的指纹。用于签署身份签名的证书将由对方的身份提供程序进行验证。如果通过全部检查，就表示对等连接通过了身份验证，由身份提供程序断言的身份也可以显示为经过验证的身份。

请注意，这种身份断言是由浏览器生成并检查的。无论是对于下载到本地的 JavaScript 还是服务器代码本身，Web 服务器都不参与上述过程。Web 服务器唯一能做的可能是阻止身份签名交换。请注意，Web 服务器也可能会提供身份指示，此指示与身份提供程序所提供的身份不一定匹配。

显然，如果用户希望在会话中保护隐私，或者不希望自己的身份在多个 WebRTC 站点间联系起来，则应该不启用此身份服务。

到目前为止，Mozilla 对身份提供程序机制表现出了极大的兴趣，很可能会推出第一款支持它的浏览器。根据它的实用性和 WebRTC 开发人员的采用情况，其他浏览器未来可能会增加对它的支持。

有关身份提供程序 API 的信息，请参见 8.3.1.5 节。

13.5 企业问题

企业通常采用边界安全来控制流经他们边界的 Internet 通信。对于常规的 IP 流量，通常采用防火墙。防火墙是针对 IP 访问的策略实施点。最简单的访问规则是单向过滤。一般来说，除非从 Internet 传入的数据包是为了响应自企业内部发送的请求，否则便加以阻止。此外，针对特定的 IP 地址、端口和传输，还可以设置其他规则。本地和远程 IP 地址、本地和远程端口，以及传输协议（例如，UDP 或 TCP）通常称作“五元组”，代表可以用来描述特定协议特征的五个数据点。有些协议始终使用特定的端口，从而可以对特定协议应用五元组规则。例如，Web 浏览通常使用 TCP 端口 80，因此通过在防火墙中开启端口 80，便可以启用 Web 浏览，而阻止端口 80 则可以限制 Web 浏览。诸如由 WebRTC、SIP 或 Jingle 建立的媒体流不使用熟知或已注册的端口。因此，不能简单地通过五元组过滤规则来允许或阻止这些媒体流。

对于穿过防火墙的特定应用程序遍历，可以使用应用程序层网关（Application Layer Gateway，ALG）。对于 Internet 通信，可以使用名为会话边界控制器（Session Border Controller，SBC）的专用 ALG。通常，它们使用信令通道（例如 SIP 或 Jingle）对产生的媒体流（例如 RTP 或 SRTP）进行授权。SBC 通常放置在防火墙旁的可信连接中，此连接名为 DMZ（隔离区，原指边境周围用来阻止对立的军事双方直接接触的缓冲地带）。

SBC 使用信令通道来应用策略。例如，根据目标 IP 地址或 URL，可以允许或阻止特定类型的媒体通信。此外，还可以应用其他策略，例如，是否对媒体进行录制或归档。

就 WebRTC 来说，没有标准化的信令通道，因此，除非使用标准化的信令协议，否则诸如 SBC 等设备的功能通常是有限的。即使 WebRTC 应用程序在 JavaScript 内使用信令协议，SBC 也不一定能够访问它，因为它可能通过安全 HTTP 或安全 WebSocket 连接来传输，

并与进出企业的其他各种网络通信混合在一起。

有关企业 WebRTC 问题的更深度讨论，请参见 [IEEE-COMS]。

13.6 隐私

在 WebRTC 中，有很多方面与隐私相关。身份隐私、IP 地址隐私和浏览器指纹识别都十分重要。[draft-iab-privacyconsiderations] 中包含 Internet 协议的实用指南。

13.6.1 身份隐私

在 WebRTC 中，身份隐私意味着用户可以访问网站，可以使用 WebRTC，而不会导致自己的身份遭到泄露。通常，采用 WebRTC 的网站将会在服务中使用并共享身份信息。不过，网站可能会在使用 WebRTC 时承诺匿名或专用通信。当然，尽管 IP 地址可以提供某种程度的身份信息，但网站需要知道个人的身份才能共享这些通信。当用户登录网站后，便没有协议或 API 方法可用来实施匿名或专用 WebRTC 会话。这类似于 PSTN，电话网络始终知道呼叫方的身份。虽然用户可以选择阻止提供自己的身份，但并非总能如此，例如，拨打免费（8xx）电话时，无论呼叫方如何指示，都会提供他们的身份。

此外，比较显而易见的是，如果建立专用或匿名 WebRTC 会话，则应该不会用到身份提供程序。

身份信息也可能会在信令通道中遭到泄露。例如，SIP INVITE 请求可能包含用户的非匿名 SIP URI。

13.6.2 IP 地址隐私

Internet 隐私是一个非常复杂的问题，就连 Web 浏览方面的隐私问题也非常复杂。本书不探讨诸如 Web Cookie 等话题。不过，我们这里将会讨论受 WebRTC 影响的一个隐私方面：IP 地址隐私。

每次浏览器连接至 Web 服务器时，Web 服务器都知道浏览器的 IP 地址。在大多数情况下，浏览器实际上只是通过最外层 NAT 的公共 IP 地址进行连接。例如，在图 3.7 中，无论通过手机还是平版电脑访问网页，Web 服务器都将获取 IP 地址 203.0.113.4，而不是可以区分手机和平板电脑的个体 IP 地址。但是，此 IP 地址可以透露很多与用户相关的信息，例如位置信息，此类信息通常用于显示特定于位置的广告和搜索结果。有些服务允许浏览器连接至服务器，而不会透露这些信息。洋葱路由器（The Onion Router，TOR）[TOR] 就是这样一个示例，这是一种匿名浏览服务。不过，一般来说，Web 服务器始终知道与之连接的浏览器的 IP 地址，而且可以对此信息进行各种处理，包括记录、存储或与任何人共享。例如，有些站点会显示这些信息以用于故障排除目的 [WHATSMYIP]，或者用来关联匿名帖子。WebRTC 没有更改大家所熟知的任何此类 Web 特性。

但是，如果使用 WebRTC 来建立浏览器到浏览器的直接媒体流，则每个浏览器都可以了解对方浏览器的 IP 地址。即使未建立会话或对方拒绝会话，这也有可能会发生。如果将此地址作为打洞候选地址进行共享，则这些信息还可能包括内部 IP 地址（即私有 IP 地址）。这是 WebRTC 带来的一种新的隐私暴露问题，可能会被用来获取与其他用户相关的新信息。

如果隐私在 WebRTC 应用程序中十分重要，则也许可以通过几种方法来改善隐私保护：

1）浏览器可以只发送一个 IP 地址候选，即 TURN 服务器的地址，如 9.2 节中所述。这可以让另一端的浏览器了解 TURN 服务器的位置，但无法识别使用 TURN 服务器的实际浏览器。

2）可以使用虚拟专用网络（Virtual Private Network，VPN），仅将 VPN IP 地址作为候选项进行共享。

3）浏览器可以选择不将私有 IP 地址作为候选项进行共享。浏览器将会转而共享公共 IP 地址，这可能是服务提供商或企业的 IP 地址，但不会透露背后的个别用户或计算机。

4）Web 服务器可以采用专门中继所有媒体通信的策略，例如，使用 TURN 服务器。不过，用户必须信任网站能够代为处理。

在信令通道中，IP 地址信息可能会在信令消息的内容或信令消息的传输中遭到泄露。例如，对于那些使用 WebSocket 代理实施 WebRTC 的浏览器，WebSocket 代理将会识别浏览器的 IP 地址。由于 SDP 对象将包含 ICE IP 地址候选项，因此这也会提供 IP 地址信息。这还会提供私有 IP 地址信息，这些信息通常不为 Web 服务器所知，后者正常情况下只会看到外部 NAT 的 IP 地址。

需要指出的是，无论在任何 Web 浏览情形下，都必须同时与 Web 浏览器及对方 Web 浏览器共享某个 IP 地址。唯一的问题是共享哪一个 IP 地址，以及这会透露多少与用户相关的信息。

13.6.3 浏览器指纹识别

浏览器指纹识别是尝试通过特定浏览器的功能和特性来识别它的过程。网站能够识别的浏览器信息越多，指纹识别就越容易。例如，指纹识别可用于绕开浏览器的 Cookie 策略来执行跟踪。EFF 使用在线测试工具 [PANOPTICLICK] 对 Web 浏览器的独特性进行了调查。在发布的报告 [EFF-FINGER] 中，他们列出了浏览器指纹识别的首要来源：插件、字体、浏览器、接受的方法、屏幕分辨率、时区，以及启用的 Cookie。W3C 全体大会专门就此主题 [FINGERPRINT] 进行了一次讨论。

WebRTC 可以提供更多可用于指纹识别的浏览器相关信息，例如，支持的媒体类型、编解码器、网络接口，以及摄像头、麦克风和扬声器的标签等。

13.7 基于数据通道的 ZRTP

ZRTP 协议 [RFC6189] 可以通过数据通道运行，以便确认媒体会话或数据通道连接中

没有中间人（Man-in-the-Middle，MitM）攻击者 [draft-johnston-rtcweb-zrtp]。图 13.8 显示了这一机制。就本质而言，这种方法取代了身份提供程序等第三方，来验证用于建立 DTLS 连接的自签名证书的指纹。实际上，在建立 DTLS 连接后，将会建立一个数据通道，然后通过数据通道建立 ZRTP 会话。在这个过程中，将会比较 DTLS 指纹，如果指纹匹配，就表明没有 MitM 攻击者。

通过比较每个浏览器上显示的短身份验证字符串（Short Authentication String，SAS），用户可以确认没有针对 ZRTP 的 MitM 攻击。

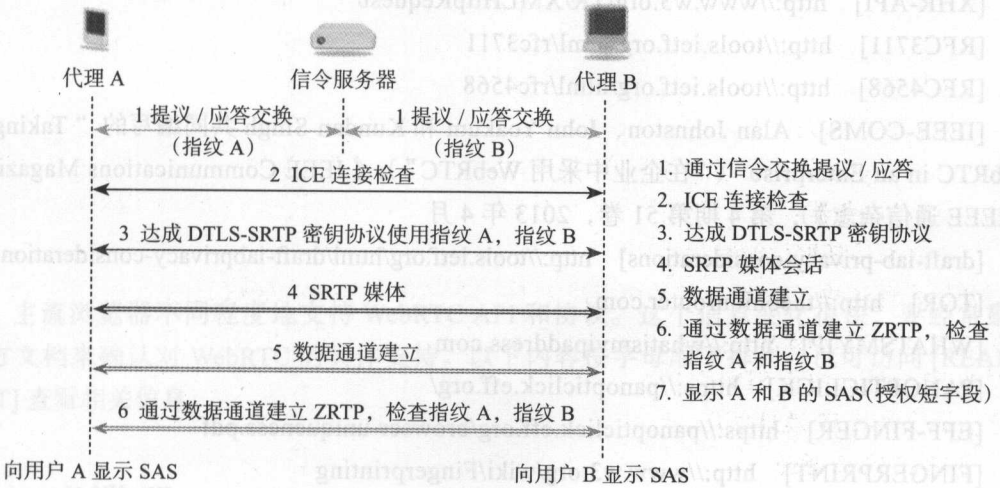


图 13.8 使用 ZRTP 为媒体和数据通道提供 MitM 保护

13.8 总结

WebRTC 安全是一个非常重要的课题，随着这项新技术的普及，将会获得大量的关注。WebRTC 可以利用现有的 Web 安全特性，例如，安全 HTTP。此外，它还具备媒体路径中内置的安全功能。通过身份提供程序机制，它还可以使用浏览器身份方法。

WebRTC 浏览器必须支持自动更新，以便用户获得最新的修补程序和更新。

13.9 参考资料

[draft-ietf-rtcweb-security]
http://tools.ietf.org/html/draft-ietf-rtcwebsecurity
[draft-ietf-rtcweb-security-arch] http://tools.ietf.org/html/draft-ietfrtcweb-security-arch
[APPLIED-CRYPTO] http://www.amazon.com/Applied-Cryptography-Protocols-

Algorithms-Source/dp/0471117099

[HTTPS-EVERY] https://www.eff.org/https-everywhere

[DANE] http://www.internetsociety.org/deploy360/resources/dane/

[DNSSEC] http://tools.ietf.org/html/rfc4033

[VOIP-SEC] http://www.amazon.com/Understanding-Security-Artech-Telecommunications-Library/dp/1596930500

[CORS] http://www.w3.org/TR/cors/

[XHR-API] http://www.w3.org/TR/XMLHttpRequest/

[RFC3711] http://tools.ietf.org/html/rfc3711

[RFC4568] http://tools.ietf.org/html/rfc4568

[IEEE-COMS] Alan Johnston、John Yoakum 和 Kundan Singh 共同编写的“Taking on WebRTC in an Enterprise”(“在企业中采用 WebRTC”),《IEEE Communications Magazine》(《IEEE 通信杂志》),第 4 期第 51 卷,2013 年 4 月

[draft-iab-privacy-considerations] http://tools.ietf.org/html/draft-iabprivacy-considerations

[TOR] http://theonionrouter.com

[WHATSMYIP] http://whatismyipaddress.com

[PANOPTICLICK] https://panopticlick.eff.org/

[EFF-FINGER] https://panopticlick.eff.org/browser-uniqueness.pdf

[FINGERPRINT] http://www.w3.org/wiki/Fingerprinting

[RFC6189] http://tools.ietf.org/html/rfc6189

[draft-johnston-rtcweb-zrtp] http://tools.ietf.org/html/draft-johnstonrtcweb-zrtp

第 14 章 Chapter 14

实现和应用

主流浏览器不同程度地支持 WebRTC API 和协议。这个信息变化很快，要经常翻阅官方文档来确认对 WebRTC 的具体支持。以下内容按字母顺序排列，也可访问 [READY-YET] 查看相关信息。

14.1 浏览器

14.1.1 Apple Safari

暂不支持。注意到 iOS 平台上也正在使用 Google 的 SDK，借助 Objective C API 编写 WebRTC 应用程序。

14.1.2 Google Chrome

支持 `RTCPeerConnection`、`getUserMedia()`、`MediaStreams` 和 `DataChannels`，在标准 Chrome [CHROME] 浏览器中可用。同时支持 VP8、Opus、STUN 和 TURN。安卓平台上的 Google Chrome 浏览器 [CR-ANDROID] 同样支持 WebRTC。

Chrome 浏览器为 WebRTC 开发者提供了许多有用工具，这些工具在 Tools/Developer 菜单下。同时，`chrome://webrtc-internals` 提供了丰富的调试信息，包括 JavaScript 状态和媒体信息等。

更多细节，包括最新的 Chrome 浏览器支持 WebRTC 的情况，可以访问 http://www.webrtc.org/chrome。

14.1.3 Mozilla Firefox

Firefox [FIREFOX] 默认支持 `RTCPeerConnection`、`getUserMedia()`、`MediaStreams` 和 `DataChannels`。Firefox 支持 VP8、Opus 和 STUN，但不支持 TURN。关于 Firefox 对 WebRTC 支持的最新情况，参见 http://developer.mozilla.org/en-US/docs/WebRTC。安卓平台上的 Firefox 24 同样支持 WebRTC。Firefox OS [FIREFOX-OS] 也将支持 WebRTC。

14.1.4 Microsoft Internet Explorer

暂不支持。然而，类似 webrtc4all [WEBRTC4ALL] 这样的项目旨在提供一种临时的解决方案。值得注意的是，[WEBRTC4ALL] 使用的 API 前缀有别于 Chrome 和 Firefox，并不在 adapter.js 库中。因此，本书中的演示应用未经修改并不适用。

微软发布过几个版本的替代 API 文档 [CU-RTC-WEB]。关于是否将这些 API 集成到现有标准中的讨论还在继续。由此可见，在 IE 浏览器中是否会支持 WebRTC，在多大程度上支持，现在并不明确。

14.1.5 Opera

Opera 20 [OPERA] 默认支持 `RTCPeerConnection`、`getUserMedia()`、`MediaStreams` 和 `DataChannels`。与 Chrome 浏览器一样，Opera 也是基于 Chromium 浏览器开发的。然而，截止到本书出版时，Opera 浏览器并未百分百支持本书中的演示程序。

14.2 其他浏览器

Google 把 Chrome WebRTC 的实现构建成可下载，可嵌入的 SDK。许多非浏览器设备有机会借助于这个 SDK 使用 WebRTC。

在 WebRTC 之上有大批的 API 实现。EasyRTC [EASYRTC] 是一个包含了 Web 应用、服务器组件、代码片段等的工具包，它的初衷是帮助人们更方便地使用 WebRTC，即使标准本身还在不断更新。Open RTC（ORTC）API [ORCA] 在诞生之初被视为 WebRTC 的竞争者，但现在它的开发致力于如何能方便地基于 WebRTC 1.0 构建 ORTC API，或者能够原生地支持。ORTC 试图绕开在 WebRTC 中用到的提议 / 应答模型。值得注意的是，这项工作目前由 W3C 的一个社区工作组负责，并不是一个标准化草案制订组。

还有其他很多 WebRTC 相关的通信 API 提供商。值得一提的是 Phono，Phono [PHONO] 在面世之初是一组让用户能方便创建 SIP 软电话的 API，很快演变为通信界的 jQuery，同时提供 jQuery 和 JavaScript 实现，让使用者能轻而易举地与服务器或另一个 Phono 终端建立通信。在 WebRTC 出现之前，Phono 支持通过 JavaScript 和 Adobe Flash 插件获取设备麦克风，现在也加入了对 WebRTC API 的支持。

14.3 STUN 和 TURN 服务器实现

rfc5766-turnserver [TURNSERVER] 是一个出色的基于标准的开源 STUN 和 TURN 服务器。本书中的演示程序正是使用它的 TURN 服务器。搭建和配置 TURN 服务器的详细步骤可参考 http://webrcbook.com/turnserver 。

14.4 参考资料

[READY-YET] http://iswebrtcreadyyet.com/

[WEBRTC4ALL] http://code.google.com/p/webrtc4all

[CHROME] http://www.google.com/chrome

[CHR-ANDROID] http://www.google.com/intl/en/chrome/browser/mobile/android.html

[FIREFOX] http://www.firefox.com

[FF-OS] http://developer.mozilla.org/en-US/Firefox_OS

[CU-RTC-WEB] http://lists.w3.org/Archives/Public/public-webrtc/2012Oct/att-0076/realtime-media.html

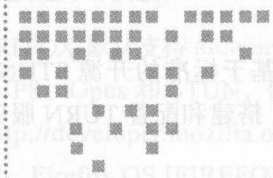
[OPERA] http://www.opera.com

[EASYRTC] http://www.easyrtc.com

[ORCA] http://www.w3.org/community/orca/

[PHONO] http://phono.com

[TURNSERVER] http://code.google.com/p/rfc5766-turn-server/



Appendix A 附录 A

W3C 标准流程

目前，万维网联盟正在开发用于 WebRTC 的标准 API。除了 WEBRTC 工作组，还有许多其他 W3C 工作组和任务组致力于 WebRTC 工作。

A.1 万维网联盟简介

万维网联盟（World Wide Web Consortium，W3C）[W3C] 是由 HTML 的创造者 Tim Berners-Lee 建立的，旨在促进 HTML 和其他 Web 技术的发展。随着时间的推移，W3C 逐步设立了一套流程，成为严格意义上的标准开发组织（Standards Development Organization，SDO），其中包括致力于达成共识，设立纠纷调解流程，处理非会员的评论，开展可行性测试，以及永久存储和提供关于规范的讨论内容、决议和官方发展草案。W3C 是一个会员制组织，只有会员能够设立工作组，而且部分工作组仅限会员参加会议和意见征集活动。然而，所有官方文件和最终的讨论结果都在 Web 上公开发布，不收取任何费用。WEBRTC 工作组中的所有讨论都是公开的。

虽然 W3C 会采取多种团体结构来讨论规范事宜，但最终只由一个工作组给出官方认可的 W3C 标准，称为“ W3C 推荐标准”。推荐标准当然会以 HTML 格式发布。W3C 工作组按活动领域进行组织，结构上较为松散，但这主要是为了方便 W3C 的内部管理，对于小组的工作方向影响很小。此外，还设有一些任务组，负责在两个或多个工作组之间进行非正式合作。媒体任务组就是一个例子，它负责制定“媒体捕获和流”文档。

在 W3C 标准的制定过程中，规范开发阶段和对共识的关注是两个最重要的因素。W3C 中的规范（又称为“技术报告”）将会经历以下阶段而逐步成熟：

1）初步公开工作草案。

2）工作草案。

3）最终工作草案（LCWD）。

4）候选推荐标准。

5）建议推荐标准。

6）推荐标准。

前三个阶段共同表明，技术报告在技术层面尚未完成，而且尚不代表广泛的共识。后三个阶段共同表明，规范在技术层面已制定完毕，而且经过广泛审核。

具体而言，规范最初以初步公开工作草案的形式公布于众。在此阶段，没有达成任何共识，只是各方承诺开发这一知识财产。接下来，随着关于规范中技术内容的探讨不断深入，该工作组会定期发布数量不定的工作草案。虽然高效率、高效能的小组会在制定这些工作草案时就计划凝聚共识，但从正式角度而言，工作草案不代表共识，只能称为“尚在拟定的工作”。

一旦工作组认为规范已经在技术层面制定完毕，就会将此文档发布为最终工作草案（Last Call Working Draft，LCWD）。有趣的是，其他 W3C 小组通常会在此时首次审核该草案，因此在进入下一阶段之前，往往会出现不止一个 LCWD。此时，还必须开始记录有关每项公开评论的处理方式。只有当针对每项评论的处理都达到评论人的满意之后，才能将规范推进到下一阶段（特殊情况除外）。通过 LCWD 阶段之后，下一步是制定“候选推荐标准”文档，其中包含详细的实施报告计划。该计划一般会列出各种主张和测试，以及对于每项功能，需要实施多少次才能进入下一阶段。此阶段的目的不在于测试实施情况，而是确保通过足够多的实施结果来确认规范是可实施的，并且在理想情况下，这些实施结果能够进行互操作。除解释和编辑变更之外，对于规范的任何更改都将导致文档返回工作草案阶段；同样，这是为了确保得到充分的审核并取得共识。在根据计划获得充足的实施结果后，如果无需进行重大变更，技术报告可进入倒数第二阶段，即建议推荐标准。此阶段基本上是一种形式，实际上已超出工作组的工作范围。假设此阶段（通常为一个月）内没有公开的异议，报告将自动进入推荐标准阶段。

在关于 W3C 流程的讨论中，如果没有谈及共识，那么讨论就是不充分的。W3C 非常重视共识要求，要求尽一切可能解决每一个异议。虽然 IETF 仅要求取得大体的共识即可，但 W3C 要求取得全体共识。这样会增加制定推荐标准所需的时间，但降低了明知此文档会背离大批目标用户还加以推行的概率。

A.2 W3C WEBRTC 工作组

W3C 中负责 WebRTC 事务的主要工作组是 WEBRTC 工作组 [WEBRTCWG]。虽然 IETF RTCWEB 和 W3C WEBRTC 小组是同时开始设立的，但后者所用的时间却远长于前

者。这里的问题在于商定要使用哪个文档作为初始文档。WHATWG [WHATWG] 是一个独立组织，致力于对 HTML 本身制定修订内容，籍此为 HTML 的开发方向提供建议。其中一项修订是在现有基础上进行扩展，以便在两个浏览器之间建立媒体连接（又称为“对等连接”）。这需要大量的时间来解决版权问题，但最终 W3C 制定了版权声明，允许 WEBRTC 工作组使用 WHATWG 的 PeerConnection 文本作为小组工作的起点。

A.3 WEBRTC 与其他 W3C 工作组的关系

WebRTC 的目标是定义用于在浏览器间建立直接媒体连接的 API，而不是定义什么是媒体，近端或远端将会或可能如何使用媒体，或者媒体与 HTML 的现有功能有何关联。由于需要选择同步或异步传输机制，因此 WebRTC 确实需要定义媒体同步的某些方面，但上面列出的其他方面将由 W3C 内部的其他小组处理。下面列出了部分相关的小组。

媒体捕获任务组 [MEDIAWG]——媒体捕获任务组由以下两个 W3C 工作组的成员组成：WEBRTC API 工作组和设备 API 工作组。此小组的目标是联合定义 getUserMedia()，此 API 调用可用于请求本地媒体（访问摄像头、麦克风、扬声器等）。另外，此小组还负责定义 MediaStream 接口的内核，因为这对于两个小组都意义重大。

HTML 工作组 [HTMLWG]——显然，HTML 工作组侧重于超文本标记语言的开发，该语言是万维网的基础语言。虽然 WEBRTC 工作组与 HTML 工作组之间没有直接的合作关系，但 WEBRTC 小组中的许多参与者都是 HTML 工作组中活跃的参与者或追随者。更重要的是，WebRTC 参与者了解，WebRTC API 必须与现有的 HTML API 和标记一致并良好集成。目前，HTML 的最新版本是 HTML5。

音频工作组 [AUDIOWG]——音频工作组负责开发用于在 HTML 内部进行更高级音频操作的 API。虽然 WEBRTC 工作组与音频工作组之间没有直接的联系，但音频工作组拥有影响 MediaStream 和 getUserMedia 接口（由媒体捕获任务组定义）的用例。

A.4 参考资料

[W3C] http://www.w3c.org

[WEBRTCWG] http://www.w3.org/2011/04/webrtc

[WHATWG] http://www.whatwg.org

[MEDIAWG] http://www.w3.org/wiki/Media_Capture

[HTMLWG] http://www.w3.org/html/wg

[AUDIOWG] http://www.w3.org/2011/audio

附录 B Appendix B

IETF 标准流程

Internet 工程任务组（Internet Engineering Task Force，IETF）负责制定 WebRTC 的标准协议。除了 RTCWEB 工作组，还有许多其他 IETF 工作组致力于 WebRTC 工作。

B.1 Internet 工程任务组简介

Internet 工程任务组（IETF）[IETF] 是负责制定 Internet 标准协议的国际标准机构。迄今为止，IETF 已制定了 IP、TCP、UDP、DNS、SIP、RTP、HTTP 和 SMTP 等常用的标准化协议。IETF 将其标准文档发布为带编号的序列文档，称为“意见征求书”（Request for Comments，RFC）。请注意，并非所有 RFC 都是 IETF 文档。另外，并非所有 IETF RFC 都是标准文档。在最终定稿并批准为 RFC 之前，标准文档的工作草案被称为“Internet 草案”。

IETF 的内部工作主要使用邮件列表通过电子邮件完成。每个工作组都有单独的邮件列表。IETF 中有关 WebRTC 的大部分工作都由 RTCWEB 工作组中讨论，但相关工作也会在其他工作组中进行。

工作组不收取会员费，任何人都可以通过订阅邮件列表、发送评论、撰写 Internet 草案或参加面对面的 IETF 会议，为这份工作贡献自己的力量。具体工作按领域（称为“工作组”）进行组织。

下面列出了制定 IETF 文档的常规流程步骤：

1）提交个人 Internet 草案。

2）采用工作组文档。

3）工作组最后征集意见（Working Group Last Call，WGLC）。

4）IETF 最后征集意见。

5）IESG 批准为 RFC。

Internet 草案是通过电子邮件或 IETF 网站上的在线表格提交给 IETF 的工作文档。Internet 草案必须符合特定的格式要求，并包含知识产权和版权声明。Internet 草案频繁更新，如不更新或没有确定为 RFC，则会在 6 个月后自动过期。初始版本为 -00（从零开始计数），每次更新后，版本号都会递增。作为个人草案，作者可以在其中写入自己喜欢的任何内容。Internet 草案由文件名标识，始终以“draft-lastname-wgname”开头。其中，“lastname”是主要作者或编辑的姓氏，“wgname”是可能讨论此工作的工作组的名称。文件名的剩余部分是带连字符号的标题或内容。例如，draft-burnett-rtcweb-constraints-registry 是 Daniel C. Burnett 为 RTCWEB 工作组撰写的有关约束注册表的个人 Internet 草案。

由于 WebRTC 工作尚未结束，因此本书中讨论的许多文档都是 Internet 草案，而且其中的内容可能已经改变。本书中的超链接会自动将你定向到最新版本。不过，其文档名称可能已经改变，也可能是多个文档已合并在一起，亦可能是一个文档分解成了多个文档。

IETF 中的工作组都经过特许，可根据特定的协议里程碑目标来制定文档。工作组首先会“采用”一个草案作为起点，然后努力制定出达成共识的文档，以实现特定的里程碑目标。从此刻开始，该草案的作者或编辑应该尽力在草案中反映工作组的共识。当草案经过修订后，文件名将舍去作者名称，更改为“draft-ietf-wgname”。由于文件名已更改，因此版本将重置为 -00。工作组文档往往会得到更广泛的审核，并被纳入面对面 IETF 会议的议程，并列在工作组页面和摘要上。

一旦工作组主席认为，Internet 草案已完成并代表小组的共识，他们会举行工作组最后征集意见（WGLC）活动，进行最终审核和评价。如果最终做出重大变更或更新，则可能要针对此文档再举行多次 WGLC 活动。当此过程完成并且主席认为草案取得“大体共识”后，他们会将文档推进到 IETF 最后征集意见阶段，在整个 IETF 内进行最终审核。当最终审核完成之后，Internet 工程任务组指导委员会（Internet Engineering Task Force Steering Committee，IESG）的成员会举行投票。如果成功通过投票表决，此 Internet 草案将得到批准并进入 RFC 编辑器的队列。几个月后，此草案将获得一个 RFC 编号并发布为 RFC。

IETF 发布的 RFC 分为多种类型。最常见的是建议标准（Proposed Standard，PS）和信息文档。建议标准实际就是 IETF 协议标准。信息 RFC 不定义协议或标准，而是定义文档要求、问题或协议背后的动机。某些 WebRTC 文档会作为信息 RFC 发布，但大多数会作为建议标准发布。

B.2 IETF RTCWEB 工作组

IETF 中负责 WebRTC 工作的主要工作组是 RTCWEB 工作组 [RTCWEB WG]，其中 RTCWEB 是实时通信 Web（Real-Time Communications Web）的缩写。然而，WebRTC 工

作涵盖很多领域，因此有许多工作组都参与到此项工作中。除了 Internet 草案（工作标准文档），WebRTC 还参考其他 IETF RFC（意见征求书，即制定完毕的标准文档）。本书中也列出并介绍了这两种文档。

B.3 RTCWEB 与其他 IETF 工作组的关系

除了 RTCWEB 工作组中的工作，还有一些与 WebRTC 相关的现行工作在下面列出的其他工作组中完成。

AVTCORE [AVTCOREWG]——音频视频传输核心工作组（Audio Video Transport Core Working Group，AVTCORE）致力于为 WebRTC 使用的实时传输协议（Real-time Transport Protocol，RTP）制定标准扩展项。此小组负责定义如何同步并一起发送不同类型的媒体。

MMUSIC [MMUSICWG]——多方多媒体会话控制工作组（Multiparty Multimedia Session Control Working Group，MMUSIC）致力于为 WebRTC 使用的会话描述协议（Session Description Protocol，SDP）制定标准扩展项。

RMCAT [RMCATWG]——RTP 媒体拥塞避免技术工作组（RTP Media Congestion Avoidance Techniques Working Group，RMCAT）致力于为通过 UDP 传输的 RTP 媒体开发拥塞控制技术。目前，此小组正在制定相关要求、RTP 或 RTCP 数据包中的反馈信息以及拥塞控制算法。WebRTC 将使用这些方法来确保 WebRTC 的广泛使用不会导致 Internet 格外拥堵，让 WebRTC 媒体会话远离拥塞，从而为用户提供最佳体验。

TRAM [TRAMWG]——TURN 修订和现代化工作组（TURN Revised And Modernized Working Group，TRAM）致力于开发 STUN 和 TURN 扩展项，其中许多都与 WebRTC 相关。它们涉及新的传输机制、身份验证方法和常规扩展项。

B.4 参考资料

[IETF] http://www.ietf.org

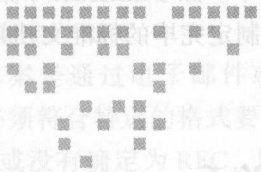
[RTCWEBWG] http://tools.ietf.org/wg/rtcweb

[AVTCOREWG] http://tools.ietf.org/wg/avtcore

[MMUSICWG] http://tools.ietf.org/wg/mmusic

[RMCATWG] http://tools.ietf.org/wg/rmcat

[TRAMWG] http://tools.ietf.org/wg/tram



Appendix C 附录 C

术 语 表

ABNF——扩增的巴科斯诺尔范式（Augmented Backus-Naur Format）。这种元语言用于定义基于文本的 Internet 协议（例如 SDP 和 URL）的语法。它最初在 RFC 822 中定义，目前最新的规范是 RFC 5234。

API——应用程序编程接口（Application Programming Interface）。API 是软件组件用于彼此通信的接口。

HTML5——万维网上的网页和应用程序使用的超文本标记语言的最新版本。HTML 最初使用 XML 定义简单的标记标签。目前，HTML5 支持层叠样式表（Cascading Style Sheets，CSS），并支持 JavaScript 等脚本语言。WebRTC 是 HTML5 的组成部分，用于处理浏览器中的实时语音流、视频流和数据流。

JavaScript——网页上使用的解释脚本编程语言。尽管它的名称包含 Java，但实际两者相差很大。目前，最高级的网页和应用程序都使用 JavaScript。从技术上讲，JavaScript 是 ECMA-262（ECMAScript）标准的一种实施方案。实际上，JavaScript 和 ECMAScript 术语可互换使用。

Jingle——这是一种多媒体信令协议，对可扩展消息传递和联机状态协议（Extensible Messaging and Presence Protocol，XMPP）（RFC 6120，又称为“Jabber”）进行了扩展。Jingle 是通过 XEP-0166 定义的。Jingle 使用 RTP/SRTP 进行媒体传输和 ICE NAT 遍历，并支持媒体信息到 SDP 的映射。

NAT——网络地址转换（Network Address Translation，NAT）。NAT 功能通常内置在 Internet 路由器或集线器中，用于将一个 IP 地址空间映射到另一个空间。通常，人们使用 NAT 来让多部设备共享同一 IP 地址，例如家用路由器或集线器中的 NAT 就经常用于此目的。此外，企业或服务提供商还使用 NAT 来划分 IP 网段，以简化控制和管理。许多

Internet 协议，尤其是使用 TCP 传输或客户端 / 服务器体系结构的协议，都可以轻易遍历 NAT。但是，对等协议和使用 UDP 传输的协议就会遇到巨大的困难。WebRTC 中的 NAT 遍历使用 ICE 协议。 有关 NAT 和打洞工作原理的详情，请参见《 SIP: Understanding the Session Initiation Protocol 》(《 SIP：了解会话初始协议》) 第 3 版的第 10 章。NAT 有时也用于指代网络地址转换器。

提议 / 应答——媒体协商是指通信会话中的双方（例如两个浏览器）进行通信并就可接受的媒体会话达成一致的过程。“提议 / 应答”是一种媒体协商方式：首先，一方向另一方发送其支持并要设置的媒体类型和功能，这称为“提议”；随后，另一方予以回应，指示在所提议的媒体类型和功能中，哪些是此会话支持并可以接受的，这称为“应答”。为建立和修改会话，可能会多次重复此过程。提议 / 应答是一般性的术语，但在 WebRTC 中时，它通常指代定义提议 / 应答协议（SIP 使用 SDP 的一种方式）的 RFC 3264。要了解如何在 WebRTC 中通过交换 SDP 会话描述来协商媒体会话，就必须研究提议 / 应答。有关 SDP 提议 / 应答的示例，请参见《 SIP: Understanding the Session Initiation Protocol 》(《 SIP ： 了解会话初始协议》) 第 3 版的第 13 章。

Node.js——WebRTC API 在 Web 浏览器中的客户端侧运行。它们属于 JavaScript API，可利用内置在浏览器中的事件循环来处理 Web GUI 的标志性异步输入和输出。WebRTC 有许多方法在执行时比较耗时，因而采用以下设计：完成时执行应用程序开发人员提供的回调。Node.js 通常简称为“ Node”，同时提供 JavaScript 解释器和事件循环功能，并包含若干用于联网和文件访问的模块以及一个便捷的模块 / 数据包管理系统。对于通过 PHP、Perl 或其他语言处理的每个请求，传统的 Web 服务器会产生一个新进程。这对于孤立的页面而言并没有问题，但对于各请求彼此关联（甚至连在一起）但仍然异步的服务而言，将难以协调正在运行的各个请求处理程序，需要设置共享的数据库、共享的文件系统或显式的进程间通信。另一方面，Node 的 http 服务器使用单一进程线程处理所有传入的请求。每个请求都要排队，并通过事件循环处理。因此，在 Node 中编写 Web 服务器就类似于在浏览器中编写动态网页——每个代码片段都由一个事件或回调启动，并且所有代码必须确保其不产生阻塞。此编程方式要求的好处在于，所有请求都在相同的内存和进程空间内处理，因而可以大幅简化针对多个请求的控制与同步。在处理包含 WebRTC API 的网页时，通过 Node 可方便地处理页面请求和 XHR 推送 / 请求，并能够对使用 Node 作为浏览器之间的信令网关的各个浏览器维护浏览器状态。

对等连接——此术语用于指代两个“对等端”（在 WebRTC 上下文中，指两个 Web 浏览器）之间为传输音频、视频和数据而建立的直接连接。此类连接通过使用 RTCPeerConnection 和相关 API 建立。

SIP——会话初始协议（Session Initiation Protocol，SIP）。SIP 是应用程序级别的信令协议，适用于 Internet 通信、IP 语音和视频。SIP 由 RFC 3261 定义，并使用提议 / 应答协议所定义的 SDP 会话描述。

WebSocket——WebSocket 协议用于在 Web 浏览器和 Web 服务器（由浏览器使用 HTTP 连通）之间建立长期有效的双向 TCP 连接。

Appendix D 附录 D

补充阅读和信息资源

http://webrtc.org.cn

最具权威的中文互联网实时通讯社区，也是本书中文版官方网站。

可扫描下方二维码关注微信公众号：

有关 HTML5 和 JavaScript 的背景信息，建议阅读以下网站上易于理解的教程：

http://www.w3schools.com

有关会话初始协议等 Internet 通信信令协议的背景信息，建议阅读：

Alan B. Johnston 编写的《SIP: Understanding the Session Initiation Protocol》（《SIP：了解会话初始协议》），Artech House，波士顿，2009 年，283 页，第 3 版，ISBN-13:978-1607839958。该书还讨论了 NAT 遍历和打洞、SDP 会话描述以及 SDP 提议 / 应答。

Henry Sinnreich 和 Alan B. Johnston 合著的《Internet Communications using SIP: Delivering VoIP and Multimedia Services with Session Initiation Protocol》，John Wiley and Sons，纽约，2005 年，298 页，第 2 版。ISBN-13:978-0471776574。

有关 RTP 和媒体传输的背景信息，建议阅读：

Colin Perkins 编写的《RTP:Audio and Video for the Internet》，Addison-Wesley Professional，纽约，2003 年，432 页，ISBN-13:978-0672322495。

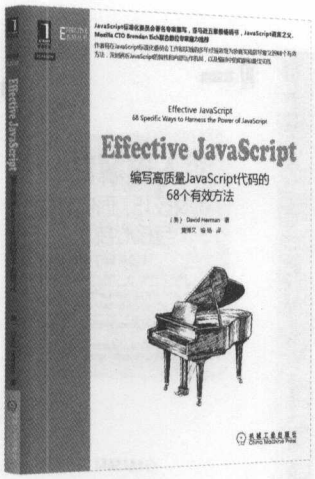
有关 VoIP 和视频的 Internet 通信安全，建议阅读：

Alan B. Johnston 和 D. Piscitello 合著的《Understanding Voice over IP Security》，Artech House，波士顿，2006，276 页，ISBN-13:978-1596930506。

有关娱乐性的虚构网络犯罪和黑客故事（同时也教授了计算机和 Internet 安全的基础知识），建议阅读：

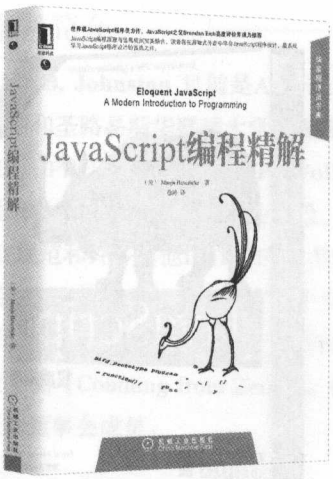
Alan B. Johnston 编写的《Counting from Zero》，2011 年，281 页，ISBN-13:978-1461064886 或 Kindle 电子书。

推荐阅读



Effective JavaScript: 编写高质量JavaScript代码的68个有效方法

作者：David Herman ISBN：978-7-111-44623-1 定价：49.00元



JavaScript编程精解

作者：Marijn Haverbeke ISBN：978-7-111-39665-9 定价：49.00元

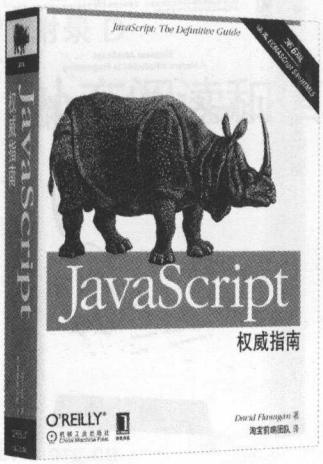
HTML5 WebSocket权威指南

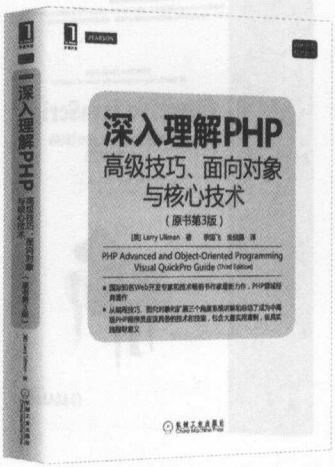
作者：Vanessa Wang 等 ISBN：978-7-111-45641-4 定价：49.00元

构建实时Web应用：基于HTML5 WebSocket、PHP和jQuery

作者：Jason Lengstorf 等 ISBN：978-7-111-43983-7 定价：69.00元

推荐阅读





JavaScript权威指南（原书第6版）

从1996年以来，本书已经成为 JavaScript 程序员的《圣经》。

第6版涵盖HTML5和ECMAScript 5。很多章节完全重写，以便跟得上当今的最佳Web开发实践。该版本的新增章节描述了jQuery和服务器端JavaScript。

对于那些希望学习Web编程语言的有经验的程序员和希望精通JavaScript的当前JavaScript程序员，本书最适合不过了。

深入理解PHP：高级技巧、面向对象与核心技术（原书第3版）

国际知名Web开发专家和技术畅销书作家最新力作，PHP领域经典著作。

从编程技巧、面向对象和扩展三个角度系统讲解和总结了成为中高级PHP程序员应该具备的技术和技能，包含大量实用案例，极具实践指导意义。

如果你已经具备一定的PHP编程基础，想使开发效率更高，想把应用做得更好，那么这本书应该是你需要阅读的。本书旨在为想修炼成为高级PHP程序员的中初级PHP程序员提供实用的方法和建议。

作者简介

Alan B. Johnston 目前是Avaya公司的杰出工程师和圣路易斯华盛顿大学的兼职讲师。他在SIP、IP语音（Voice over IP，VoIP）和互联网通信领域有超过13年的从业经验，参与编写了SIP规范和许多其他IETF RFC，其中包括ZRTP媒体安全协议。他著有四本关于互联网通信、SIP和安全的技术类畅销书，还有一本科技惊险小说《Counting from Zero》。他还是SIP论坛的董事会成员。

Daniel C. Burnett 目前是Tropo的首席科学家和Voxeo（Aspect旗下的一家公司）的标准总监。他从事计算机标准工作十余年，曾编写和编辑了W3C的许多标准，这些标准为当今的大多数自动化交互式语音应答（Interactive Voice Response, IVR）系统奠定了基础。由于在自动语音识别（声音辨别）领域的标准制定方面贡献卓越，他曾两度荣获《Speech Tech》杂志颁发的久负盛名的“语音杰出人物”奖。作为PeerConnection和getUserMedia W3C WEBRTC规范的编辑以及IETF的参与者，Daniel从一开始就投入到了这个令人振奋的新领域。

封面设计 · 王建敏

WebRTC

APIs and RTCWEB Protocols of the HTML5 Real-Time Web, Third Edition

WebRTC权威指南

（原书第3版）

中国互联网发展迅猛。基础云服务、开源技术、HTML5、移动SDK等技术，让中国的开发者能最快速地开发移动和网页App，与世界比肩。下一个风口，一定会是融合了实时通信技术的应用，比如视频通话、实时互动直播等。WebRTC无疑是推动实时通信技术发展至关重要的技术之一。

—— 赵斌（Tony Zhao），声网Agora.io CEO

2011年，Google公司正式对外推出了开源实时通信项目WebRTC。在之后的几年里，WebRTC成为一个全球性的平台，为全球700多个公司和项目提供创造性地融合了实时通信技术的产品和服务。我们非常高兴中国的开发者终于有了中文的WebRTC权威指南，帮助他们进行WebRTC的实践与开发。

—— Google WebRTC Team

本书是互联网实时通信开发者和技术决策者的权威参考指南。作者Daniel C. Burnett博士是WebRTC标准的主要作者，在书中对标准的方方面面做了精确到位的介绍。Alan Johnston 博士则是今日通信业核心标准SIP的主要作者，多年的行业实践和全局视野让他能够深入浅出地给出WebRTC相关技术问题和发展方向的真知灼见。

本书循序渐进地介绍了WebRTC，阐述了诸如本地媒体、信令等概念，并通过独立可运行的演示程序来介绍对等连接。此外，还详细描述了浏览器媒体协商过程，如何使用Wireshark来监控WebRTC协议的注意事项以及例子捕捉等。书中给出了大量示例代码、各类数据和图表，所有的代码都可以在http://webrtcbook.com/code3.html免费下载，你还可以在http://demo.webrtcbook.com上试用。

上架指导：计算机/网络通信

ISBN 978-7-111-54715-0

9 787111 547150 >

定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.
网上购书：www.chi .com
数字阅读：www.hzmedia.com.cn